

STATUS OF THESIS

Title of thesis **Separator Database and SPM-Tree Framework for Mining Sequential Patterns using PrefixSpan with Pseudoprojection**

I **DHANY SAPUTRA** hereby allow my thesis to be placed at the Information Resource Center (IRC) of Universiti Teknologi PETRONAS (UTP) with the following conditions:

1. The thesis becomes the property of UTP
2. The IRC of UTP may make copies of the thesis for academic purposes only
3. This thesis classified as

☐ Confidential

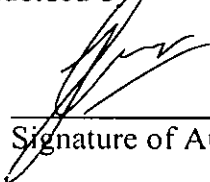
☒ Non-confidential

If the thesis is confidential, please state the reason:

The contents of the thesis will remain confidential for ____ years.

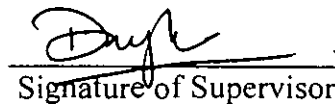
Remarks on disclosure:

Endorsed by


Signature of Author

Jalan Hayam Wuruk B-11
Jombang
Jawa Timur – 61411
Indonesia

Date: JUNE 30, 2008


Signature of Supervisor

Department of Computer and Information Sciences
Universiti Teknologi PETRONAS
Bandar Seri Iskandar, Perak
Malaysia

Date: June 30, 2008

APPROVAL

UNIVERSITI TEKNOLOGI PETRONAS

Approval by Supervisors

The Undersigned certify that they have read, and recommend to The Postgraduate Studies Programme for acceptance, a thesis entitled "Separator Database and SPM-Tree Framework for Mining Sequential Patterns Using PrefixSpan with Pseudoprojection" submitted by Dhany Saputra for the fulfillment of the requirements for the degree of Master of Science in Computer Information and Science.

Date

Signature

:

Dyfl

Main Supervisor

:

Dr. Dayang Rohaya Awang Rambli

Date

:

June 30, 2008

Signature

:

FOONG OI MEAN

Co-Supervisor

:

FOONG OI MEAN

Date

:

June 30, 2008

UNIVERSITI TEKNOLOGI PETRONAS

Separator Database and SPM-Tree Framework for Mining Sequential Patterns

Using PrefixSpan with Pseudoprojection

By

Dhany Saputra

A THESIS

SUBMITTED TO THE POSTGRADUATE STUDIES PROGRAMME
AS A REQUIREMENT FOR THE DEGREE OF MASTER OF SCIENCE
IN COMPUTER INFORMATION AND SCIENCE PROGRAMME

BANDAR SERI ISKANDAR

PERAK

APRIL 2008

DECLARATION

I hereby declare that the thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTP or other institutions.

Signature :  _____

Name : DHANY SAPUTRA

Date : JUNE 30, 2008

DEDICATION

*This thesis is dedicated to my beloved father and mother
for their prayer and encouragement.*

ACKNOWLEDGEMENT

All praise is for Allah Almighty, Most Merciful, Most Compassionate, and may His blessing, peace, and favors descend in perpetuity on our beloved Prophet Muhammad SAW, who is a mercy for all the worlds.

To my family: Thank you my father and my mother, for your endless support, for your care, and for your love all the time. Also thanks to my two younger sisters, Nila and Elsa.

To UTP academic staffs: I would like to express my gratitude and appreciation to my main supervisor, Dr Dayang Rohaya Awang Rambli, not only towards my accomplishment to this work but also towards very careful reading on my thesis, continuous guidance, support, and valuable advices throughout my study here in UTP. I am also thankful and appreciative to my co-supervisor, Ms. Foong Oi Mean, for her considerate and conscious guidance for the completion of this research work. I would also like to express my gratitude to Prof. Dr. Jubair Jwamear A.S. Al-Jaafar, Dr. Etienne Schneider, Dr. Mohamed Nordin Zakaria, and Ms. Yong Suet Peng for brightening my mind with their expertise and insightful discussions during the completion of this research.

To the university, staffs and technicians: I am grateful to Computer and Information Sciences Department and Postgraduate Studies Programme in Universiti Teknologi PETRONAS, which have provided all necessary supports including my presentation on international conference held in Kuching as well as publishing three papers and one international journal. I wish to thank Mr Rasky (VR Lab Technician), Ms Azarina (Multimedia Lab IV and V Technician), and Encik Muhamad (PG Office Workstation 1.6 Technician) for their help on providing me access and privilege to the computers for research; Encik Fadil Ariff, Kak Norma, Kak Aida, and all other as postgraduate

office staffs for their superb benevolence on helping me with the postgraduate research administration matters.

To my partners in research: I would like to express my special gratitude to Pak Dani Adhipta and Pak Agus Arif for helping and guiding me through the thesis writing process; Pak Ayok, Om Arief Rahman, and Pak Bayu Erfianto for being my supportive research discussion partners; extraordinary appreciation to Barkah Yusuf for being my great sensei on Data Mining; great gratefulness to Dr Jiawei Han (University of Illinois), Prof. Dr. Stephan Olariu (University of Old Dominion), and Mr. Febriliyan Samopa (Ph.D. candidate at Hiroshima University, Japan) for their kind comments and helpful inputs to my research. Also to all of IT PG Students in Block 2: Aini and Intan (thank you for your great help to translate my abstract into BM), Pak Jati, Pak Totok, Pak Bambang Ariwahjoedi, and Andri Kusbiantoro (thank you for helping me with spiritual support on research).

Tremendous gratitude to all of my supportive UTP postgraduate friends especially citizen of V5C (Pak Balzach, Pak Eko, Pak Rofiq, Pak Lava, Pak Hudiyo, Mas Dedi BS Uhuy, Pak Agus, Pak Budi Agung, Mang Rulli, Pak Oyas, Pak Wardo), citizen of V4 and V3 (Dewa132, Pak Agni, Rizal, Faisal, Fauzan, Acong, Kiki, Amelia, and Afny), citizen of Bandar U (Bang Adi, Farhat, Mas Agung Dewandaru, Mas Irfan, Mas Rahmat, Mbak Erna, Pak Totok, Pak Ridho, and Bu Ayok), and PG Students partner-in-crimes, especially on makan-makan and die-hard member of DCFC-girls (Chin, Ari, Diyana, Shu, Sha, Mas, Fadziella, Fatimah and Rashidah). I really enjoyed every single moment I have spent with you. You are all unquestionably great friends to me.

ABSTRACT

Sequential pattern mining is a new branch of data mining science that solves inter-transaction pattern mining problems. Efficiency and scalability on mining complete set of patterns is the challenge of sequential pattern mining. A comprehensive performance study has been reported that PrefixSpan, one of the sequential pattern mining algorithms, outperforms GSP, SPADE, as well as FreeSpan in most cases, and PrefixSpan integrated with pseudoprojection technique is the fastest among those tested algorithms. Nevertheless, Pseudoprojection technique, which requires maintaining and visiting the in-memory sequence database frequently until all patterns are found, consumes a considerable amount of memory space and induces the algorithm to undertake many redundant and unnecessary checks to this copy of original database into memory when the candidate patterns are examined. Moreover, improper management of intermediate databases may adversely affect the execution time and memory utilization. In the present work, Separator Database is proposed to improve PrefixSpan with pseudoprojection through early removal of uneconomical in-memory sequence database, whilst SPM-Tree Framework is proposed to build the intermediate databases. By means of procedures for building index set of longer patterns using Separator Database, some procedure in accordance to in-memory sequence database can be removed, thus most of the memory space can be released and some obliteration of redundant checks to in-memory sequence database reduce the execution time. By storing intermediate databases into SPM-Tree Framework, the sequence database can be stored into memory and the index set may be built. Using Java as a case study, a series of experiment was conducted to select a suitable API class named `Collections` for this framework. The experimental results show that Separator Database always improves, exponentially in some cases, PrefixSpan with pseudoprojection. The results also show that in Java, `ArrayList` is the most suitable choice for storing `Object` and `ArrayIntList` is the most suitable choice

for storing integer data. This novel approach of integrating Separator Database and SPM-Tree Framework using these choices of Java Collections outperforms PrefixSpan with pseudoprojection in terms of CPU performance and memory utilization. Future research includes exploring the use of Separator Database in PrefixSpan with pseudoprojection to improve mining generalized sequential patterns, particularly in handling mining constrained sequential patterns.

ABSTRAK

Perlombongan pola berjujukan adalah penemuan baru dalam sains perlombongan data yang diaplikasikan bagi mengenal pasti bentuk perlombongan pola antara transaksi. Cabaran bagi aplikasi perlombongan pola berjujukan ini adalah ketepatan dan keberkesanannya untuk mengenal pasti himpunan pola dalam perlombongan itu. Beberapa kajian yang telah dibuat menunjukkan PrefixSpan, penambahbaikkan GSP, SPADE, dan juga FreeSpan dapat memperbaiki kepantasan algoritma dalam perlombongan pola berjujukan ini. Namun, dengan pengenalan persepaduan PrefixSpan dan teknik pseudoprojection, kajian telah menunjukkan bahawa ia merupakan algoritma yang terpanjang dalam perlombongan pola berjujukan. Walau bagaimanapun, teknik pseudoprojection memerlukan data disimpan dalam memori dan datanya perlu sentiasa dipantau sehingga kesemua pola dikenalpasti; memerlukan jumlah ruang memori yang besar di samping memerlukan algoritma PrefixSpan melakukan pemeriksaan yang berulang dan berlebihan. Pemeriksaan ini dibuat ke atas salinan data yang telah diambil dari pangkalan data asal ke dalam memori komputer ketika proses bentuk pola-pola dikenalpasti. Tambahan pula, pengurusan yang tidak sempurna ketika proses perantaraan data di antara pangkalan data boleh menyebabkan tempoh masa yang diambil adalah lama dan ia juga memerlukan penggunaan ruang memori yang besar. Justeru, bagi kajian ini Separator Database dicadangkan untuk penambahbaikkan penggunaan sediaada persepaduan PrefixSpan dengan teknik pseudoprojection. Menerusi kaedah ini, data yang tidak ekonomis yang telah disalin ke dalam memori dari pangkalan data akan dinafikan seawal mungkin. Sementara itu, SPM-Tree Framework juga diketengahkan dalam membina proses perantaraan data di antara pangkalan data. Dengan penggunaan prosedur untuk membina himpunan indeks bagi pola yang panjang menerusi Separator Database, beberapa prosedur bagi data yang telah disalin ke dalam memori dari pangkalan data dapat dikecualikan. Justeru, penggunaan kapasiti ruang di dalam memori dapat dikurangkan di samping mengurangkan pengulangan pemeriksaan ke atas memori di dalam pangkalan data

dan ini sekaligus mengurangkan tempoh masa pemprosesan yang diperuntukan. Di samping itu, kemasukan pangkalan data perantara ke dalam SPM-Tree Framework, membolehkan pangkalan data berjujukan disimpan di dalam memori dan himpunan indek dapat dibina. Dengan menggunakan Java dalam kajian kes, beberapa siri eksperimen telah dijalankan untuk memilih satu kelas API bernama `Collections` yang sesuai untuk rangka kerja ini. Hasil dari beberapa siri eksperimen menunjukkan bahawa `Separator Database` sentiasa menyumbang kepada penambahbaikan persepaduan `PrefixSpan` dengan teknik `pseudoprojection` dan peningkatannya adalah pesat dalam kebanyakan kes. Hasil kajian tersebut juga telah menunjukkan, dalam Java; `ArrayList` adalah pilihan yang paling sesuai untuk menyimpan `Object`, dan `ArrayIntList` adalah pilihan yang paling sesuai untuk penyimpanan data integer. Pendekatan baru yang menyepadukan `Separator Database` dan SPM-Tree Framework ini, jelas menunjukkan penambahbaikan yang positif ke atas penyepaduan `PrefixSpan` dengan teknik `pseudoprojection` dari aspek prestasi CPU dan penggunaan memori.

TABLE OF CONTENTS

STATUS OF THESIS.....	i
APPROVAL	ii
DECLARATION	iv
DEDICATION.....	v
ACKNOWLEDGEMENT	vi
ABSTRACT.....	viii
ABSTRAK.....	x
TABLE OF CONTENTS.....	xii
LIST OF TABLES.....	xiv
LIST OF FIGURES	xv
 CHAPTER 1 INTRODUCTION	 1
1.1 Background	1
1.2 Problem Statement	3
1.3 Research Aim and Contributions.....	4
1.4 Scope and Limitations.....	5
1.5 Thesis Organization	7
 CHAPTER 2 LITERATURE REVIEW	 8
2.1 Sequential Pattern Mining.....	8
2.1.1 Definition of Terms.....	9
2.2 Approaches to Mining Sequential Patterns.....	12
2.2.1 Generate and Test Approach (Apriori Approach)	12
2.2.1.1 AprioriAll, AprioriSome, DynamicSome	13
2.2.1.2 GSP	16
2.2.1.3 PSP	19
2.2.1.4 Discussion	20
2.2.2 Vertical Format Approach	20
2.2.3 Pattern Growth Approach	22
2.2.3.1 FreeSpan	25
2.2.3.2 PrefixSpan.....	27
2.2.3.3 PrefixSpan with Pseudoprojection Technique	33
2.2.3.4 MEMISP	37
2.2.3.5 Discussion	38
 CHAPTER 3 IMPROVEMENTS ON PREFIXSPAN WITH PSEUDOPROJECTION	 42
3.1 SPM-Tree Framework	42
3.1.1 The Need for a Framework	42
3.1.2 Definition of Terms.....	43
3.1.3 The SPM-Tree Framework	44

3.1.4	Data Structure Requirements for The Framework.....	46
3.1.5	Data Structure Candidates for The Framework	48
3.2	Separator Database.....	49
3.2.1	Background	49
3.2.2	Definition of Terms.....	50
3.2.3	The Separator Database	51
3.2.4	Effects of Separator Database to Several Database Characteristics	54
CHAPTER 4 RESULTS AND DISCUSSION.....		56
4.1	Selection of Suitable Data Structure for SPM-Tree Framework	56
4.1.1	CPU Performance of Appending Integers into Different List Collections	57
4.1.2	CPU Performance of Retrieving Integers into Different List Collections	58
4.1.3	Memory Usage of Appending Integers into Different List Collections	60
4.1.4	Memory Usage of Retrieving Integers into Different List Collections.....	61
4.1.5	CPU Performance of Appending Objects into Different List Collections	62
4.1.6	CPU Performance of Retrieving Objects into Different List Collections	63
4.1.7	Discussion.....	63
4.2	Performance and Scalability Test of Separator Database	64
4.2.1	CPU Performance versus Minimum Support	65
4.2.2	CPU Performance versus Dataset Density.....	67
4.2.3	CPU Performance versus Dataset Size	68
4.2.4	CPU Performance versus Transaction Length.....	69
4.2.5	Memory Usage versus Transaction Length	70
4.2.6	Discussion	71
CHAPTER 5 CONCLUSIONS AND FUTURE WORKS.....		74
5.1	Conclusions.....	74
5.2	Future Works	75
BIBLIOGRAPHY.....		76
APPENDIX A: EXPERIMENT DATA		80

LIST OF TABLES

Table I: Example of Transactional Database	10
Table II. Sequence Database Version	11
Table III. Database projection and the set of sequential patterns	31
Table IV. Index set for 1-patterns	35
Table V: Index Set for n-patterns ($n > 1$)	36
Table VI. Separator Database	52
Table VII. Execution time of Appending Integers (in second)	80
Table VIII. Execution time of Retrieving Integers (in second)	80
Table IX. Memory Usage of Appending Integers (in MB)	81
Table X. Memory Usage of Appending and Retrieving Integers (in MB)	81
Table XI. Execution Time of Appending Objects (in second)	82
Table XII. Execution Time of Retrieving Objects (in second)	82
Table XIII. Execution Time versus Database Density (in second)	83
Table XIV. Execution Time versus Database Size (in second)	83
Table XV. Execution Time versus Minimum Support Threshold (in second)	83
Table XVI. Execution Time versus Average Transaction Length (in second)	84
Table XVII. Memory Usage versus Average Transaction Length (in MB)	84

LIST OF FIGURES

Figure 1. Approaches to Mining Sequential Patterns	12
Figure 2. PrefixSpan with Pseudoprojection	29
Figure 3. <i>SPM-Tree Framework</i>	44
Figure 4. CPU performance of appending integers into different List Collections	57
Figure 5. CPU performance of retrieving integers into different List Collections	58
Figure 6. Memory usage of appending integers into different List Collections	60
Figure 7. Memory usage of appending and retrieving integers into different List Collections	61
Figure 8. CPU performance of appending Objects into different List Collections	62
Figure 9. CPU performance of retrieving Objects into different List Collections	63
Figure 10. CPU Performance versus Minimum Support on Dataset-1	65
Figure 11. CPU Performance versus Dataset Density on Dataset-2	67
Figure 12. CPU Performance versus Dataset Size on Dataset-3	68
Figure 13. CPU Performance versus Transaction Length on Dataset-4	69
Figure 14. Memory Usage versus Transaction Length on Dataset-4.....	70

CHAPTER 1

INTRODUCTION

1.1 Background

Rapid progress in digital data acquisition and storage technology has resulted in the growth of huge databases. This amazing progress is happening in all areas such as supermarket transactional data, credit card usage records, telephone call details, government statistics, images of astronomical bodies, and molecular databases [1]. However, data collected are seldom revisited and most often important decisions are made based not on the information gained from these data but rather on a decision maker's intuition. Therefore, this progress couples with high interest of extracting valuable information to the decision makers have motivated the growth of knowledge discovery in database (KDD). In essence, knowledge discovery in databases comprises following steps [2]:

- Data cleaning (to remove noise and irrelevant data)
- Data integration (to combine multiple data sources)
- Data selection (to retrieve only relevant data to the analysis task)
- Data transformation (to transform the data into forms suitable for mining)
- Data mining (to apply intelligent methods and to extract patterns)
- Pattern evaluation (to identify the truly interesting patterns)
- Knowledge presentation (to visualize the mined knowledge to user)

The discipline of data mining focuses on extracting interesting patterns or knowledge from large information repositories. A pattern is interesting if it is easily understood by human, valid on new or test data with some degree of

certainty, potentially useful, and novel.

There are various types of data mining techniques [3], such as classification, clustering, and association rules mining. Classification builds a model that can classify a class of objects to predict the accurate classification. Clustering groups set of objects based on their similarities. Meanwhile, association rule mining, known as frequent pattern mining, extracts interesting frequent patterns. It is widely used for market basket or transaction data analysis. Association rules mining concerns with finding intra-transaction patterns, i.e. concurrent purchase pattern. Nevertheless, association rules mining is not capable of figuring out inter-transaction patterns, i.e. sequence of purchase patterns. Thus, a new branch of data mining science, named sequential patterns mining was born to solve inter-transaction patterns mining.

Sequential pattern mining plays a paramount role in many data mining tasks [4], such as mining complete sequential patterns [5], [6], [7], [8], [9], [10], [11], [12], [13], mining constrained sequential patterns [6], [14], [16], [17], [18], mining closed sequential patterns [19], [20], [21], mining approximate sequential patterns [22], [23], incremental mining sequential patterns [24], and mining structured patterns [25]. Sequential pattern mining also has broad applications, such as stock and market analysis, web log click streams analysis [26], natural disasters (e.g., earthquakes), mining sequential alarm patterns in a telecommunication database [27], DNA sequences analysis and gene structures analysis [28].

Mining complete sequential patterns or simply named sequential pattern mining has been intensively studied during recent years. Efficiency and scalability on mining complete set of sequential patterns is the challenge of sequential pattern mining. Several approaches have been proposed to cope with this challenge. They are generate-and-test approach, vertical format approach, and pattern growth approach. Earlier algorithms, such as AprioriAll

[5], GSP [6], and PSP [7], adopt apriori approach to discover sequential patterns. Then, Zaki came with an efficient SPADE algorithm [8], which proposed vertical format sequence database, instead of using apriori approach. SPADE outperforms GSP by a factor of two through experiments. Meanwhile, pattern growth approach was introduced [9] by proposing FreeSpan algorithm [10]. Still applying pattern growth approach, Jian Pei et al [11] proposed PrefixSpan algorithm and pseudoprojection technique, which outperforms SPADE, FreeSpan, and GSP [12]. Despite outperforming others, PrefixSpan still bears an inherent cost in performance due to the inefficiency of using in-memory sequence database. Therefore, the focus of this thesis is to propose further improvements on PrefixSpan with pseudoprojection.

1.2 Problem Statement

PrefixSpan with pseudoprojection builds two intermediate databases, i.e. index set (also known as pseudoprojection database) and in-memory sequence database when searching for patterns. If the data structure implementation strategy to build and use those two temporary databases is not managed carefully, then the execution time and memory utilization will be considerably affected. Thus, a framework to store the intermediate databases in PrefixSpan with pseudoprojection is an essential implementation issue to bring up to front.

PrefixSpan, though reducing recursive construction of projected database through pseudoprojection technique [11], [12], bears an inherent cost in performance due to the inefficiency of using in-memory sequence database. Pseudoprojection technique requires to maintain and to visit the in-memory sequence database frequently until all patterns are figured out. Maintaining in-memory sequence database consumes considerable amount of memory space during mining. Frequent access to in-memory sequence database is not efficient since there are many redundant and unnecessary checks to this copy

of original database into memory when the candidate patterns are examined.

1.3 Research Aim and Contributions

The overall aim of this research is to improve the performance of PrefixSpan with pseudoprojection by speeding up the mining process and reducing the memory utilization.

To cope with the highly efficient and scalable mining complete set of sequential patterns as the challenge of mining sequential patterns, this thesis improves PrefixSpan with pseudoprojection technique on the following two contributions:

- Separator Database was proposed to substitute the uneconomical in-memory sequence database in PrefixSpan with pseudoprojection technique.
- SPM-Tree Framework was proposed as a framework to build in-memory sequence database and to build index set. Using Java as a case study, it was shown that `ArrayList` is the most suitable `Collections` for storing `Object` and `ArrayIntList` is the most suitable `Collections` for storing integer data.

The work in this thesis has resulted in the following publications.

- Dhany Saputra, Dayang Rohaya Awang Rambli, Foong Oi Mean, "An Efficient Data Structure for General Tree-Like Framework in Mining Sequential Patterns using MEMISP", in *Proceedings of The 5th International Conference on Information Technology in Asia (CITA '07)*, 2007, pp. 133-139.
- Dhany Saputra, Dayang Rohaya Awang Rambli, Foong Oi Mean, "Mining Sequential Patterns using I-PrefixSpan", in *Proceedings of World Academy of Science, Engineering, and Technology, International Conference on Knowledge Mining (ICKM '07)*, 2007, pp. 499-504.

- Dhany Saputra, Dayang Rohaya Awang Rambli, Foong Oi Mean, “Mining Sequential Patterns using I-PrefixSpan”, *International Journal of Computer Science and Engineering (IJCSE '08)*, vol 2, pp. 49-54, Spring 2008.
- Dhany Saputra, Dayang Rohaya Awang Rambli, Foong Oi Mean, “Sequential Pattern Mining using PrefixSpan with Pseudoprojection and Separator Database”, in *Proceedings of World Academy of Science, Engineering, and Technology, International Conference on Knowledge Mining (ICKM '07)*, 2007, pp. 49-54.

The first publication has contributed on SPM-Tree Framework, which is covered in Chapter 3 and 4. The second and third publication have contributed on the incorporation of SPM-Tree Framework and Separator Database, which is covered in Chapter 3, while the fourth publication has contributed on more experiments and lemma justifications to Separator Database, which is covered in Chapter 3 and 4.

1.4 Scope and Limitations

This thesis focuses on mining sequential patterns of small database. The intermediate databases for mining, either the extraction from sequence dataset to in-memory sequence database or the recursive index set creation for longer patterns must be able to fit into main memory when the database is small. It is quite plausible to mine sequential patterns from large dataset using PrefixSpan with pseudoprojection and many researchers have developed this, but this thesis confines only for small database since to cope with large database, it is required to partition the in-memory sequence database and this problem needs another careful techniques and experiments.

The output is complete sequential patterns with user-specified minimum

support threshold. It is not the purpose of this thesis to study mining maximal sequential patterns, mining constrained sequential patterns, mining closed sequential patterns, mining approximate sequential patterns, incremental mining sequential patterns, and mining structured patterns. Thus, no attempt has been made here to develop mining sequential patterns with user-specified constraints such as (1) time constraints, i.e. minimum and/or maximum time gaps between adjacent itemsets in a pattern; (2) sliding windows, i.e. the new definition of time for phrase “same transaction” as specified by user; and (3) taxonomies, where putting super-category patterns is necessary.

The system is implemented on Java programming language. Programs written in Java are typically slower and require more memory than those written in natively compiled languages such as C or C++ [29]. However, the speed of programs built in Java has improved a lot due to Just In Time (JIT) compilation in 1998 for Sun Java Virtual Machine (JVM) [30] and optimizations in the JVM itself introduced over time, such as HotSpot has become the default for Sun JVM in 2000 [31]. Hence, when JIT is compiled, Java’s performance is mostly lower than performance of compiled languages (e.g. C or C++), close to other JIT compiled languages (e.g. C#), but much better than languages without an effective native-code compiler (e.g. PHP, Perl). Based on Dr. Dobb’s Journal on Microbenchmarking C++ and Java [32], while array operation performance is better in C than in Java; but 32 and 64 bits arithmetic operations, file I/O, and exception handling have similar performance to comparable C programs; and most importantly Collections, Object creation/destruction performance, and method calls are much better in Java than in C++. Moreover, as of 2006, it is estimated there are millions of Java developers [33] and many businesses have already used Java as their integrated enterprise system. Thus, this thesis chooses Java as the study case for selecting suitable Collections for SPM-Tree Framework.

1.5 Thesis Organization

The remainder of the thesis is arranged as follows.

- In Chapter 2, the thesis presents description of sequential pattern mining, definition of terms in sequential patterns mining, and discussion on three approaches in sequential pattern mining.
- In Chapter 3, the two proposed improvements of PrefixSpan with pseudoprojection, SPM-Tree Framework and Separator Database are explained. The correctness of both improvements is verified using lemmas.
- Chapter 4 shows the results of performance analysis of the five Java List Collections, which will be suitable for the proposed SPM-Tree Framework. This chapter also shows the results of performance analysis of Separator Database on small database.
- The thesis concludes in Chapter 5 and some future works are presented.

CHAPTER 2

LITERATURE REVIEW

This chapter is organized around three major topics: a general introduction to sequential pattern mining, the definition of terms in sequential pattern mining, and a review of the three main approaches to sequential pattern mining including the algorithms.

2.1 Sequential Pattern Mining

Sequential pattern mining is one of data mining functionalities having high complexity of data [5]. It is recently becoming an intensive study in data mining. Sequential pattern mining differs from frequent pattern mining. Frequent pattern mining discovers which items are bought together in a transaction [34], [35]. It is concerned with finding intra-transaction patterns, while sequential pattern mining finds inter-transaction patterns [5].

The objective of sequential pattern mining is to find all frequent sequential patterns with a user-specified minimum support [5]. User-specified minimum support threshold enables mining to prune non-interesting patterns. The discovery of sequential patterns problem was inspired by retailing industry problems. As part of market basket analyses, sequential pattern mining helps multi-item retailing business with registered customers to increase their profitability.

To illustrate the input and output of sequential pattern mining, suppose the computer shop database contains list of customers, list of transaction time of each customer, and list of items bought for each customer's transaction time.

Then “80% of customers buy computer, then earphone, and then digital camera” may be one of the worthwhile sequential patterns found. This purchasing need not be consecutive. Those customers who buy several other items in between also support this pattern.

As part of market basket analyses, sequential pattern mining helps multi-item retailing business with registered customers to increase their profitability. There might be several competitive advantages of sequential pattern mining in retailing industry:

1. The sales representatives could persuade the customers to buy the predicted subsequent item(s) the next time customers come again. Special discounts, for instance, may attract the customers to buy just then.
2. The retailing industry may accelerate the subsequent purchases chains. Thus, the sales figure as well as, for sure, profits can be obtained earlier.
3. The retailing industry could take special eyes on sequential patterns sales and consider some different strategies apart from the regular sales.

2.1.1 Definition of Terms

In this subsection, some terminologies that are used throughout the thesis are introduced. Let I be a set of all **distinct items**. An **itemset** is a non-empty subset of I , denoted as $(x_1 x_2 \dots x_m)$ where each x_i ($i = 1, 2, \dots, m$) is an item in I . If an itemset contains only one item, the bracket can be omitted. Without loss of generality, we assume that items in an itemset are sorted alphabetically. A **sequence**, is an ordered list of itemsets, denoted by $\langle s_1 s_2 \dots s_n \rangle$ where each s_j ($j = 1, 2, \dots, n$) is an itemset. An item can occur at most once in an itemset of a sequence, but it can occur multiple times in different itemsets of a sequence. A sequence with k items ($k = \sum_j |s_j|$) is called a **k -sequence**. A sequence $\alpha = \langle a_1 a_2 \dots a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 \dots b_m \rangle$,

denoted as $\alpha \sqsubseteq \beta$, if there exists integers $1 \leq j_1 < j_2 < \dots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \dots, a_n \subseteq b_{j_n}$.

A **sequence database** S is a collection of $\langle sid, s \rangle$, where sid is the unique identifier of sequence s . The **support** of any candidate pattern α in a sequence database S is defined as the total number of sequences in the database containing α , i.e.,

$$\text{support}(\alpha) = \left| \left\{ \langle sid, s \rangle \mid \langle sid, s \rangle \in S \wedge (\alpha \sqsubseteq s) \right\} \right|.$$

Given a positive integer **minimum support** min_sup as the user-specified support threshold, a candidate pattern α is called a **sequential pattern**, or simply **pattern**, in a sequence database provided that $\text{support}(\alpha) \geq min_sup$. A pattern s is **maximal** if s is not a subsequence of any other pattern. A pattern with k items is called a **k -pattern**. Sequential pattern mining finds the complete set of sequential patterns in the sequential database, given minimum support threshold. The following example illustrates these terminologies.

Table I: Example of Transactional Database

Customer ID	Transaction Time	Items Bought
1	Nov 2 '07	a
1	Nov 3 '07	a, b, c
1	Nov 9 '07	a, c
1	Nov 22 '07	d
1	Nov 25 '07	c, f
2	Nov 19 '07	a, d
2	Nov 22 '07	c
2	Nov 23 '07	b, c
2	Nov 30 '07	a, e
3	Nov 2 '07	e, f
3	Nov 27 '07	a, b
3	Nov 28 '07	d, f
3	Nov 29 '07	c
3	Nov 30 '07	b
4	Nov 5 '07	e
4	Nov 10 '07	g
4	Nov 16 '07	a, f
4	Nov 20 '07	c
4	Nov 25 '07	b
4	Nov 29 '07	c

Table II. Sequence Database Version

Sequence ID	Sequence
1	$\langle a(abc)(ac)d(cf) \rangle$
2	$\langle (ad)c(bc)(ae) \rangle$
3	$\langle (ef)(ab)(df)cb \rangle$
4	$\langle eg(af)cbc \rangle$

Example 2.1: Let our database in Table I (used as a running example throughout the thesis) be the transactional database on hand and the minimum support = 50%. The transactional database must initially be transformed into sequence database format before sequential pattern mining is started. Table II represents the transformed sequence database from Table I. This sequence database contains four sequences and the set of distinct items of this sequence database is $\{a, b, c, d, e, f, g\}$.

Consider the sequence with sequence ID 1, which is $\langle a(abc)(ac)(d)(cf) \rangle$. This sequence has five itemsets: a , (abc) , (ac) , d , and (cf) . This sequence is a 9-sequence since there are nine items forming this sequence. As $\langle a(bc)a \rangle$ is a subsequence of $\langle a(abc)(ac)(d)(cf) \rangle$, the sequence with sequence ID = 1 contributes one support count for the candidate pattern $\langle a(bc)a \rangle$.

Since only sequences with sequence ID 1 and 2 contribute candidate pattern $\langle a(bc)a \rangle$, the support of this candidate pattern is 2. Consequently, $\langle a(bc)a \rangle$ is a pattern owing that its support does not go below 50% of the total number of sequences in the sequence database. In addition, this pattern is a 4-pattern since it contains four items. This pattern is also maximal pattern in consequence that it is not a subsequence of any other patterns.

2.2 Approaches to Mining Sequential Patterns

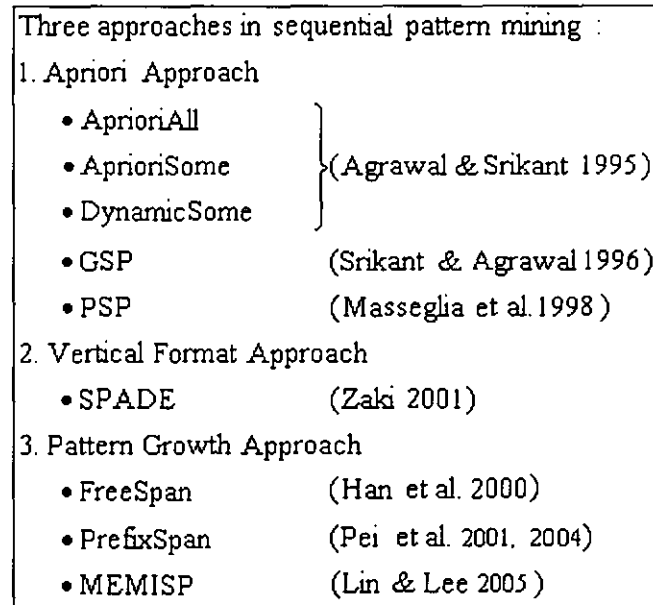


Figure 1. Approaches to Mining Sequential Patterns

In the previous decade, several approaches have been proposed to cope with highly efficient and scalable mining complete set of sequential patterns as the challenge of mining sequential patterns. Those approaches were generate-and-test (also known as apriori approach), pattern growth (also known as divide-and-conquer), and vertical format approach, as depicted in Figure 1. The subsequent subsections analyze and evaluate algorithms of those approaches.

2.2.1 Generate and Test Approach (Apriori Approach)

Most of the basic and earlier algorithms in mining sequential patterns were based on the Apriori property proposed in association rule mining, stating that *all sub-patterns of a frequent pattern are always frequent* [35]. These algorithms include AprioriAll, AprioriSome, DynamicSome, GSP, and PSP.

2.2.1.1 AprioriAll, AprioriSome, DynamicSome

In 1995, Agrawal and Srikant introduced the sequential patterns mining problem and proposed three algorithms to solve it, i.e. AprioriAll, AprioriSome, and DynamicSome [5]. These three algorithms were the first proposed algorithms to solve sequential pattern mining problem. In this work, instead of finding complete set of patterns, Agrawal and Srikant defined mining sequential patterns as finding maximal patterns amongst all sequences within user-defined minimum support threshold. These three algorithms follow similar five phases but different iteration jump and non-maximal sequence pruning.

There are five phases in mining (maximal) sequential patterns using these three algorithms, i.e. sort phase, litemset phase, transformation phase, sequence phase, and maximal phase [5]. The term “**litemset**” was introduced to mention an itemset having support no less than the minimum support threshold. In addition, in this work a k -pattern is a pattern with k itemsets instead of items. For instance, if itemset (abc) is frequent, then (abc) is a large itemset and (abc) is also considered to be large 1-pattern.

The basic concept of AprioriAll algorithm is to make multiple passes over the database. In the first pass, all litemsets substitute the seed set. Then, using the seed set on hand, the new potential large sequences, named **candidate sequences**, are generated. At the end of each pass, the support of each candidate sequence generated is counted and only candidates with minimum support are considered as the patterns and, of course, as the seed set for the next pass. The algorithm terminates when no patterns are produced at the end of a pass or when no candidate sequences could be generated.

The rest of the algorithms, AprioriSome and DynamicSome [5], comprises **forward phase**, in which one finds all patterns of certain lengths, followed by **backward phase**, in which one finds all remaining patterns. Both phases generate candidates for any passes only by patterns found on the previous pass and then both phases scan over the database to find their supports. However, DynamicSome generates candidates “on-the-fly” whereas AprioriSome generates candidates using apriori-gen function [5].

Based on those procedures, it may be concluded that AprioriAll, AprioriSome, and DynamicSome brought several optimizations to sequential patterns mining:

- **To avoid checking on every item, AprioriAll, AprioriSome, and DynamicSome map the itemsets to integers and create the transformed database.** Since non-itemsets is deleted in transformed database and itemsets containing more than one item is mapped into integers, the first three phases of these three algorithms, which are sort phase, itemset phase, and transformation phase, reduce the mining effort. If a transaction contains no itemsets, then in the transformed database the transaction is deleted. This benefits the mining especially when there are many infrequent itemsets.
- **Candidate generation ensures all possibilities to be checked.** The on-the-fly as well as the apriori-generate functions ensure all possible candidate sequences to be checked. In apriori-generate, a new candidate C_k is generated if and only if there are two patterns from previous pass L_{k-1} of whose first to $(k-2)^{\text{th}}$ itemsets are exactly the same. Furthermore, the candidate sequences need to be pruned before their supports are counted. The pruning method follows apriori property, “*all sub-patterns of a frequent pattern are always frequent*” [35]. However, on-the-fly function has no pruning, thus DynamicSome performs the worst among all these three algorithms.
- **AprioriSome focuses on mining maximal sequences.** AprioriSome avoids counting many non-maximal sequences. Thus, AprioriSome

performs better than AprioriAll especially for lower minimum supports in mining maximal sequences.

However, it may also be concluded that AprioriAll, AprioriSome, and DynamicSome draw several pitfalls:

- **Data preparation is considered as the part of mining process.** Three of the earliest phases, which are sort phase, itemset phase, and transformation phase, were not supposed to squander the total mining time and were rather supposed to be directly gone to the sequence phase with the already-prepared sequence database.
- **Multiple scan over the database is required.** If the longest sequential pattern was a k -pattern, the database scans with AprioriAll, AprioriSome, or DynamicSome k times. Mining using these three algorithms turn to be uneconomical when very long patterns existed.
- **The number of candidate sequences generated could be large.** Principally, candidate sequence set was grown from the permutation of existing itemsets, thus a truly large set of candidate sequences is generated. For instance, two itemsets $\langle(30)(40)\rangle$ and $\langle 60 \rangle$ will generate five candidates of 2-pattern: $\langle(30)(40)(30)(40)\rangle$, $\langle(30)(40)(60)\rangle$, $\langle(60)(30)(40)\rangle$, $\langle(60)(60)\rangle$, and $\langle(30)(40\ 60)\rangle$. If there were 100 frequent itemsets, these algorithms created $100 * 100 + \frac{100 * 99}{2} = 14,950$ candidates of 2-pattern.
- **AprioriSome and DynamicSome were intended only for mining maximal sequential patterns.** Both algorithms skip the discovery of several intermediate patterns. They were developed to optimize the mining maximal patterns only. Certain length to discover patterns on is decided by next function for AprioriSome and pre-specified step variable for DynamicSome [35]. If both next function and step variable were removed away for mining complete patterns, then AprioriSome and DynamicSome will work similarly with AprioriAll.

- **DynamicSome generates a much larger number of candidates in the forward phase.** On-the-fly candidate generation function causes performance degradation of DynamicSome in terms of execution time and memory utilization as this function had no pruning method. Pruning method on `apriori-generate` function cannot be applied to the forward phase of DynamicSome since a candidate C_{k+step} is generated from L_k and L_{step} , while pruning C_{k+step} need to acknowledge the frequent patterns in $C_{k+step-1}$.
- **The number of candidate sequences generated using AprioriSome can be significantly large.** A k -pattern L_k could contain candidate sequence C_k with minimum support, thus $|L_k| \leq |C_k|$, and C_{k+1} is supposed to be generated from joining L_k . However, sometimes AprioriSome uses C_k to generate C_{k+1} . If C_{k+1} is generated from joining C_k , the support counting of C_k will include some useless-to-count candidates.
- **Despite being skipped, the candidate sequences in AprioriSome are still generated and stay memory resident.** If the current iteration counter k is not equal with the leap number resulted by `next` function then the counting candidate is skipped but candidate sequences are still generated. Albeit AprioriSome skipped over counting candidates of some lengths, all candidate sequences are still generated and stay in memory.
- **AprioriAll, AprioriSome, DynamicSome fail to perform in large database.** Those three algorithms do not have any special treatment when the database is getting larger.

2.2.1.2 GSP

The following year, Srikant and Agrawal proposed GSP (Generalized Sequential Patterns) algorithm, which includes handling: (1) time constraints, i.e. minimum and/or maximum time gaps between adjacent itemsets in a pattern; (2) sliding windows, i.e. the new definition of time for phrase “same

transaction” as specified by user; and (3) taxonomies, when putting super-category patterns is necessary [6]. GSP implicitly changed the definition of mining sequential patterns, from finding maximal patterns to finding complete set of patterns.

The underlying concept of this algorithm was similar to AprioriAll, except for absence of litemset concept as well as the candidate generation procedure. GSP introduced large items, instead of large itemsets, as the seed set for the first pass. Hence, the candidate generation procedure changed a little. In GSP, the new candidates are joined from two contiguous subsequences. In the prune phase of candidate generation, candidate sequences that have a contiguous $(k-1)$ -subsequence whose support count is less than minimum support is deleted. It uses hash-tree structure [35] to reduce the number of candidates checked for a sequence and follows forward and backward phase to process the minimum/maximum gap rule and time windows filtration. It was reported in the experiments that GSP outperformed AprioriAll with on-the-fly data transformation by up to 20 times and was up to three times faster than AprioriAll with caching the transformed database into disk [6].

Based on those procedures, it may be concluded that GSP algorithm brought several benefits to sequential patterns mining:

- **GSP does not require litemset phase and transformation phase.** It is computationally expensive to transform the dataset, even on the fly. AprioriAll finds litemsets, transforms the database into mapped integer of litemset (either disk-based or on-the-fly) and uses litemsets as the seed set for second pass, while GSP uses frequent items as the seed set for the first pass instead and no data transformation is required.
- **GSP is capable to cope with time constraints, sliding windows, and taxonomies.** GSP is the generalization of the Apriori approach, which allows the mining to associate with time constraints, sliding windows, and

taxonomies. However, as explained in Section 1.5, those three constraints are not being considered in this thesis.

- **GSP reduces the candidate checking effort.** GSP employs hash-tree data structure [35]. In GSP, candidate sequences are stored in the leaves of hash-tree while the internal nodes contain hashtables. Each customer in the dataset is hashed to find the candidates contained in the associated customer sequence. By organizing candidate sequences in a hash-tree, GSP accesses the candidate sequences in efficient way.
- **GSP was the first sequential pattern mining algorithm capable to mine large databases.** In any pass k , GSP needs to store $(k-1)$ -patterns L_{k-1} , the generated candidate sequences C_k , and at least one page to buffer the database transactions. If L_{k-1} fits in memory but C_k does not, GSP generates as many candidates as will fit in the memory. The frequent sequences L_k are then written to disk, while the candidates without minimum support are deleted. If L_{k-1} does not fit in memory either, GSP generates candidates using relational merge-join techniques without any pruning. The correctness is confirmed [6], in spite of the redundant counting effort.
- **GSP counts fewer candidates than AprioriAll.** AprioriAll prunes candidate sequences by checking whether they obtained by dropping an itemset having no minimum support, while GSP checks the candidates obtained by dropping an item having no minimum support. Thus, GSP spend less time and memory space than AprioriAll.

It may also be concluded that GSP draws several pitfalls:

- **No nodes in hash-tree but leaf nodes store candidate sequences.** In hash-tree data structure, all candidates are stored in the leaves. The bulky interior nodes are merely to shows the direction to arrive to the intended leaf. It is memory waste to employ these inefficient interior nodes.
- **GSP generates a combinatorially explosive number of candidates especially in mining long sequential patterns.** For example, suppose the

longest pattern of a sequence database is of length 100. First, GSP has to generate $100 * 100 + \frac{100 * 99}{2} = 14,950$ length-2 candidates and $\binom{100}{3} = 161,700$ length-3 candidates. Without pruning, the number of length-3 candidates is $100 * 100 * 100 + 100 * 100 * 99 + \frac{100 * 99 * 98}{3 * 2} = 2,151,700$. As candidate pruning was performed on GSP, the number of candidates to be generated on each pass is $\binom{100}{n}$, where n is the length of candidates and $n > 3$. Hence, the total number of candidate sequences to be generated is roughly $\sum_{i=1}^{100} \binom{100}{i} = 2^{100} - 1 \approx 10^{30}$.

2.2.1.3 PSP

In 1998, Massegia, Cathala, and Poncelet proposed PSP, which merely introduced a novel data structure to organize candidate sequences, named **prefix-tree**. This data structure improves efficiency of candidate sequences retrieval [7]. The algorithm was quite similar with GSP.

In hash-tree structure, all candidate sequences are fully stored in the leaves. Hence, GSP traverses through the tree with the direction of interior nodes until reaching a leaf. In the correct leaf, GSP starts examining the support count. In contrast, prefix-tree structure organizes candidates according to their common items. Hence, after traversing to leaf, PSP just need to increment the support value. Moreover, after the supports of all candidates on current pass are counted, the prefix-tree is pruned to minimize the used memory space. After pruning, the tree only stores the patterns from previous passes. Nevertheless, PSP approach still does not solve the general Apriori drawbacks. It is also reported that prefix-tree structure used to store candidate sequences in PSP

requires less memory than the hash-tree structure used in GSP [7]. Thus, PSP avoids costly operations in candidate sequences verification.

2.2.1.4 Discussion

Despite those improvements, Apriori approach still draws several shortcomings due to the nature of candidate generation. Generally, Apriori approach has three major pitfalls to solve:

1. A very large set of candidate sequences could be generated since candidate sequences are built from all possible permutations of the items.
2. Apriori approach requires scanning the database as many as the length of longest pattern found.
3. Apriori approach generates a combinatorially explosive number of candidates when mining longer sequential patterns. Hence, this approach may not be efficient in mining databases having numerous and/or long patterns.

Due to these above inefficiencies of Apriori approach, some researchers started to propose other approaches analogous to ones on association rules mining, such as vertical format approach and pattern growth approach.

2.2.2 Vertical Format Approach

In 2001, Zaki proposed SPADE (Sequential Pattern Discovery using Equivalence class) algorithm, which decomposes the original problem into smaller sub-problems using equivalence classes on frequent patterns, and minimizes computational costs by using efficient searching scheme [8]. Initially, SPADE transforms the dataset into vertical format, in which each item has its own **ctid-list**, i.e. lists of pairs of customer and transaction identifiers in which the associated item occurs. Each sequence in SPADE has

a ctid-list, an array of items in the sequence, a support counter, and an integer representing the sequence-template. The main phases of SPADE include computing the frequent items and 2-patterns, the decomposition into prefix-based parent equivalence classes, and the enumeration of all other frequent sequences via breadth first search or depth first search within each class. It is reported that SPADE outperforms GSP by more than a factor of 2, and by more than an order of magnitude with precomputed support of 2-sequences.

SPADE has resulted in several improvements to sequential patterns mining:

- **SPADE scans database only three times.** Unlike Apriori approach, which needed abundant scan of database, SPADE needed only three times database scanning, one to obtain the frequent items, one to obtain 2-patterns, and the other to obtain the rest of patterns. Thus, it cuts down the I/O costs.
- **SPADE has a space-efficient join strategy to discover patterns.** SPADE uses the concept of id-list, which records the sequence ids and the transaction index positions of each frequent items found. Hence, any sequence can be obtained as a temporal join of several items. However, SPADE does not require naive production of id-lists of longer patterns by storing the transaction index positions for all items in each potential pattern. SPADE can still generate a sequence by joining the sequence's lexicographically first two $(k-1)$ length subsequences, thus SPADE reduces the use of memory space by storing only (sid,eid) pairs for any sequence.
- **SPADE has a fast join strategy to discover patterns.** The cardinalities of intermediate id-lists keep shrinking for longer patterns. As the length of discovered patterns increased, the size of its id-lists decreased, and consequently, it caused very fast temporal joins and support counting in SPADE.
- **The lattice can be decomposed into smaller pieces such that each piece can be solved independently in main memory.** One could enumerate all

frequent patterns by traversing the lattice and then performing temporal joins to obtain the support. Practically, it squanders memory space to enumerate all patterns and no intermediate id-lists will fit into memory. Nevertheless, by holding temporary id-lists for each sub-lattice, each partition can be solved independently. The supports of all patterns in each sub-lattice still can be generated using temporal joins. Then, when the main memory cannot hold the temporary id-lists of each sub-lattice, recursive sub-lattice decomposition is the solution. Concisely, depending of the main memory availability, the sub-lattices are partitioned into smaller ones, until each sub-lattice was small enough to be handled independently.

- **SPADE has excellent locality.** Hash-tree structure used in GSP has very poor locality [36] and incur the overhead of generating and searching subsequences. On the contrary, a join in SPADE requires only a linear scan of two lists, thus SPADE has excellent locality.

Despite these numerous advantages, SPADE still requires time and memory resource to convert the given database to vertical format. Once the database is converted into vertical format, SPADE is efficient in terms of memory utilization and execution time. Thus, some researchers still develop algorithms using this approach. However, other researchers started to propose algorithms exploiting pattern growth approach analogous to pattern growth approach on frequent pattern mining.

2.2.3 Pattern Growth Approach

In 2000, Jiawei Han, Jian Pei, and Yiwen Yin proposed FP-Tree (*frequent pattern tree*) structure and developed an efficient FP-Tree based mining method, FP-Growth, for mining complete set of frequent patterns by pattern

growth approach. Efficiency of mining is achieved with these techniques [37], [38]:

- A large database is compressed into a highly condensed, much smaller **FP-tree data structure**, which is constructed to store the frequent items and avoids costly, repeated database scans. Only frequent items had nodes in the tree and the tree nodes are arranged in a such way that more frequently occurring nodes have better chances of sharing nodes than less frequently occurring ones.
- **Pattern growth method** using FP-tree was developed to avoid the costly generation of a large number of candidate patterns. This method starts from a frequent items, examines only its conditional pattern base, constructs its FP-tree, and mines recursively with this FP-tree. The pattern growth is achieved via concatenation of suffix pattern with the new ones generated from a conditional FP-tree. The major operations of mining are count accumulation and prefix path count adjustment, which are usually much less costly than candidate generation and pattern matching in Apriori approach in finding frequent patterns.
- **Partitioning-based, divide-and-conquer method** was the search technique for the mining, which decomposes the mining task into a set of smaller tasks for mining confined patterns in conditional databases. Compared to Apriori's bottom-up generation of frequent itemsets combinations, it reduces the size of conditional FP-tree and conditional pattern base generated at the subsequent level. The frequent patterns mining looks for short frequent patterns then concatenates the suffix, instead of finding long ones.

All of these three techniques contribute to substantial reduction of pattern searching costs. Performance study showed that FP-Growth method is efficient and scalable for mining frequent patterns, and is about an order of magnitude faster than Apriori algorithm and also faster than TreeProjection [37].

Pattern Growth approach for efficient mining of frequent patterns without candidate generation [37], [38] inspired many research works on sequential pattern mining. FP-Growth uses FP-tree data structure to store compressed frequent patterns in transaction database and recursively mines the projected conditional FP-trees to achieve high performance. FP-tree structure explores maximal sharing of common prefix paths by reordering the items in transactions. Yet, one cannot mine sequential patterns by any sort of extensions of FP-tree structure since there is no common prefix subtree structure can be shared among them and one cannot change the order of items just to form sharable prefix subsequences. If FP-tree were used to mine sequential patterns, the FP-tree will be huge and cannot benefit mining. However, one still can explore the notion of FP-growth: divide and conquer, that is dividing the patterns to be mined based on the subsequences obtained so far and projecting the database based on the partition of such patterns. Sequential pattern mining by pattern growth approach applied this idea and there is no candidate generation in this approach. Instead of generating nonexistent candidate sequences, pattern growth approach in sequential pattern mining only grows longer sequential patterns from the shorter frequent ones. There were, basically, three algorithms applying this approach: FreeSpan [10] PrefixSpan [11], [12], and MEMISP [16].

In short, pattern growth approach has several features in general:

1. It preserves the only essential groupings of the original databases for mining, instead of generating a large number of candidates.
2. It partitions the database into database projections, instead of scanning entire database to match all candidates in each pass.
3. A substantial portion of data can be put into main memory for mining.

2.2.3.1 FreeSpan

In 2000, Jiawei Han and colleagues introduced FreeSpan, the first projection based, pattern-growth approach algorithm for mining sequential patterns [10]. It used frequent items to recursively project the database into smaller projected databases and to grow subsequences in each projected database by exploring only locally frequent fragments. FreeSpan partitions data and frequent patterns to be tested and restricts each test to the corresponding smaller projected database.

The algorithm follows these five steps:

1. Find frequent items sort by support in descending order and discard infrequent items.
2. Draw Frequent Item Matrix to generate length-2 sequential patterns and projected databases.
3. Generate a set of annotations on item-repeating patterns and annotations of projected databases, and then the matrix can be discarded.
4. Scan database one more time to generate item-repeating patterns and projected databases.
5. Recursively do matrix projection mining on projected databases if there are still longer candidate patterns to be mined.

Based on experiments conducted in the proposal, FreeSpan with bi-level projection ran faster than FreeSpan with level-by-level projection since FreeSpan with level-by-level projection requires more memory than bi-level projection and when the minimum support was below a certain point, page thrashing occurred in FreeSpan with level-by-level projection [10]. In addition, of course, FreeSpan with bi-level projection outperformed GSP on execution time and scalability.

FreeSpan has brought in several improvements to sequential patterns mining:

1. FreeSpan scanned database only three times: to find the set of frequent items, to construct the frequent item matrix, and to generate item-repeating patterns and projected databases.
2. FreeSpan need not have the sequence database residing in the main memory. Most of used memory space is only to keep the frequent item matrix. The computation of projected database can be done one by one, and mining each will require much less memory than mining the original database since the projected database size shrinks on each fragment growth.
3. FreeSpan projects and partitions the original database recursively into a set of progressively smaller intermediate databases. Consequently, the subsequent mining is confined to each projected database, relevant to a smaller set of candidates.
4. FreeSpan generated no candidate sequences. It adopted Apriori approach well but avoided generating a large number of candidates. Consequently, its search space was reduced.

Despite the significant improvements, FreeSpan's performance is limited due to the followings:

1. The k -sequence might grow at any position in projected database, thus the search for $(k+1)$ -sequence needs to check all possible permutation, and it is costly. Indeed, FreeSpan does not generate any candidate patterns, but permutation inside testing candidate patterns is still costly in terms of memory utilization and execution time.
2. Projection in FreeSpan needs to store the whole sequence from the original database without length reduction since a pattern may be generated by any substring combination in a sequence. Size of projected database in FreeSpan is still too large since the projected database still requires recording the whole sequence.
3. If a pattern appears in each sequence, its projected database does not shrink, although the removal of infrequent items helps a little bit. The main

reason is that the projected database in FreeSpan keeps the whole sequence, thus a large memory space is needed to store the projected database.

4. FreeSpan recursively constructs projected databases. In the worst case, it constructs a projected database for every pattern. FreeSpan requires excessive looping of constructing projected databases and support counting for each suspected candidate patterns until no more candidate patterns can be constructed. This recursion, to be sure, progressively requires more memory space as the patterns found is getting longer.

2.2.3.2 PrefixSpan

In 2001, Jian Pei et al proposed PrefixSpan algorithm, which solves the first abovementioned three costs of FreeSpan [11], [12] in terms of memory utilization and execution time. Those three costs of FreeSpan are the redundant checking at every possible position of potential sequences, the size of projected database, and that the projected database does not shrink when the patterns appear in each sequence. To illustrate this algorithm, the concept of prefix, suffix, projected database, and support count in projected database are introduced.

Definition 1 (Prefix). Given a sequence $\alpha = \langle a_1 a_2 \dots a_n \rangle$, $\beta = \langle b_1 b_2 \dots b_m \rangle$ ($m \leq n$) is called a prefix of α if and only if (1) $b_i = a_i$ for $i \leq m-1$; (2) $b_i \subseteq a_i$; and (3) all items in $(a_m - b_m)$ are alphabetically after those in b_m .

Definition 2 (Suffix). Given a sequence $\alpha = \langle a_1 a_2 \dots a_n \rangle$. Let $\beta = \langle a_1 a_2 \dots a_{m-1} a'_m \rangle$ ($m \leq n$) be the prefix of α . Sequence $\gamma = \langle a''_m a_{m+1} \dots a_n \rangle$ is called the suffix of α with regards to prefix β , denoted as $\gamma = \alpha / \beta$, where $a''_m = (a_m - a'_m)$. We also denote $\alpha = \beta \cdot \gamma$.

Definition 3 (Projected database). Let α be a pattern in a sequence database. The α -projected database is the collection of suffixes in the sequence database with regards to prefix α .

Definition 4 (Support count in projected database). Let β be a sequence with prefix α . The support count of β in α -projected database is the number of sequences γ in the α -projected database such that $\beta \sqsubseteq \alpha \cdot \gamma$.

In a nutshell, PrefixSpan (either with Pseudoprojection or without Pseudoprojection) comprises three phases as illustrated in Figure 2.

1. *Extraction Phase*

In this phase, the sequence database transformed from transactional database is extracted entirely into main memory. Such sequence database in memory is called as *in-memory sequence database*. In this phase, the distinct items are also obtained.

2. *Initial Mining Phase*

At first, this phase selects 1-patterns from the list of distinct items. Only distinct items within minimum support are considered as 1-patterns. Let n be the number of 1-pattern found. Next, the search space into n partitions is divided according to n 1-patterns as the prefixes. These partitions are going to be mined by initially constructing the corresponding projected databases. Hence, n projected databases exist for each prefix at the end of this phase, and the size of each projected database is less or equal to the size of sequence database.

3. *Deeper Mining Phase*

This phase produces all patterns other than 1-patterns. The existing projected database is mined recursively. Only suffix of sequences prefixed with the first

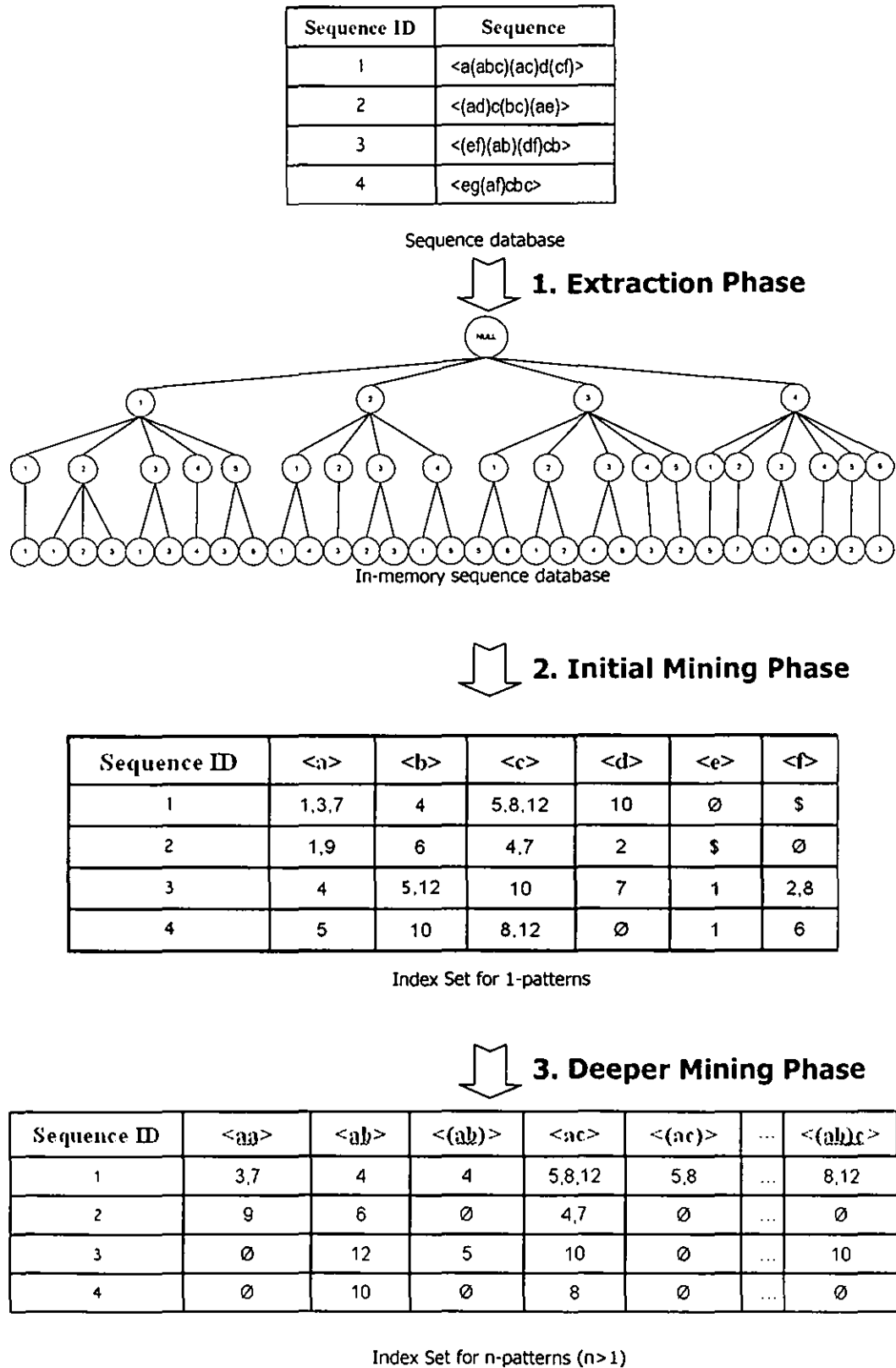


Figure 2. PrefixSpan with Pseudoprojection

occurrence of corresponding prefix should be considered. A candidate k -pattern ($k > 1$) is formed by either appending or assembling the current pattern to the candidate item, where current pattern is any $(k-1)$ pattern and the possible candidate item is all 1-patterns. Only candidate patterns within minimum support are considered as sequential patterns. Then the projected databases are built according to yielded patterns and these processes repeat until there are no patterns found in the current projected database.

Below is the example of mining sequential patterns using PrefixSpan.

Example 2.2: Given the sequence database in Table II with minimum support 50%, The distinct items are a, b, c, d, e, f , and g . Since the support of candidate pattern g is 1, the 1-patterns are $\langle a \rangle:4$, $\langle b \rangle:4$, $\langle c \rangle:4$, $\langle d \rangle:3$, $\langle e \rangle:3$, and $\langle f \rangle:3$, where the notation “ $\langle \text{pattern} \rangle: \text{count}$ ” represents the pattern and its corresponding support count.

Next, the search space is divided into six partitions, i.e. the one with prefix $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, $\langle d \rangle$, $\langle e \rangle$, and $\langle f \rangle$. For partition with prefix $\langle a \rangle$, initially build the $\langle a \rangle$ -projected database which consists of four suffixes, i.e. $\langle (abc)(ac)d(cf) \rangle$, $\langle (_d)c(bc)(ae) \rangle$, $\langle (_b)(df)cb \rangle$, and $\langle (_f)cbc \rangle$, where $(_d)$ means that the last item of prefix, that is a , together with d , are in the same itemset. From this $\langle a \rangle$ -projected database, one can discover six local frequent items, which is items within minimum support, they are $a:2$, $b:4$, $_b:2$, $c:4$, $d:2$, and $f:2$. Thus all 2-patterns prefixed with $\langle a \rangle$ are found, and they are $\langle aa \rangle=2$, $\langle ab \rangle=4$, $\langle (ab) \rangle=2$, $\langle ac \rangle=4$, $\langle ad \rangle=2$, and $\langle af \rangle=2$. Recursively, all sequential patterns with prefix $\langle a \rangle$ can be divided into six partitions, i.e. the one with prefix $\langle aa \rangle$, $\langle ab \rangle$, $\langle (ab) \rangle$, $\langle ac \rangle$, $\langle ad \rangle$, and $\langle af \rangle$.

For partition with prefix $\langle aa \rangle$, initially build the $\langle aa \rangle$ -projected database which consists of two suffixes, i.e. $\langle (_bc)(ac)d(cf) \rangle$ and $\langle (_e) \rangle$. As there are no patterns found from $\langle aa \rangle$ -projected database, the mining recursion of $\langle aa \rangle$ branch terminates.

For partition with prefix $\langle ab \rangle$, the $\langle ab \rangle$ -projected database consists of three suffix sequences, i.e. $\langle (_c)(ac)d(cf) \rangle$, $\langle (_c)a \rangle$, and $\langle c \rangle$. Recursive mining on $\langle ab \rangle$ -projected database $\langle ab \rangle$ returns four sequential patterns : $\langle (_c) \rangle$, $\langle (_c)a \rangle$, $\langle a \rangle$, and $\langle c \rangle$. Thus, patterns found from $\langle ab \rangle$ branch are $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, and $\langle abc \rangle$.

The $\langle (ab) \rangle$ -projected database consists of only two sequences, i.e. $\langle (_c)(ac)d(cf) \rangle$ and $\langle (df)cb \rangle$, which leads to the finding of four patterns $\langle c \rangle$, $\langle d \rangle$, $\langle f \rangle$, and $\langle dc \rangle$. Thus, patterns found are $\langle (ab)c \rangle$, $\langle (ab)d \rangle$, $\langle (ab)f \rangle$, and $\langle (ab)dc \rangle$.

Likewise, the $\langle ac \rangle$ -, $\langle ad \rangle$ -, $\langle af \rangle$ -, $\langle b \rangle$ -, $\langle c \rangle$ -, $\langle d \rangle$ -, $\langle e \rangle$ -, and $\langle f \rangle$ -projected databases are build in the same way. The set of sequential patterns found are shown in Table III below.

Table III. Database projection and the set of sequential patterns

Prefix	Projected database	Sequential patterns
$\langle a \rangle$	$\langle (abc)(ac)d(cf) \rangle$, $\langle (_d)c(bc)(ae) \rangle$, $\langle (_b)(df)cb \rangle$, $\langle (_f)cbc \rangle$	$\langle a \rangle:4$, $\langle aa \rangle:2$, $\langle ab \rangle:4$, $\langle aba \rangle:2$, $\langle abc \rangle:2$, $\langle a(bc) \rangle:2$, $\langle a(bc)a \rangle:2$, $\langle ac \rangle:4$, $\langle aca \rangle:2$, $\langle acb \rangle:3$, $\langle acc \rangle:3$, $\langle ad \rangle:2$, $\langle adc \rangle:2$, $\langle af \rangle:2$, $\langle (ab) \rangle:2$, $\langle (ab)c \rangle:2$, $\langle (ab)d \rangle:2$, $\langle (ab)dc \rangle:2$, $\langle (ab)f \rangle:2$
$\langle b \rangle$	$\langle (_c)(ac)d(cf) \rangle$, $\langle (_c)(ae) \rangle$, $\langle (df)cb \rangle$, $\langle c \rangle$	$\langle b \rangle$, $\langle ba \rangle:2$, $\langle bc \rangle:3$, $\langle bd \rangle:2$, $\langle bdc \rangle:2$, $\langle bf \rangle:2$, $\langle (bc) \rangle:2$, $\langle (bc)a \rangle:2$
$\langle c \rangle$	$\langle (ac)d(cf) \rangle$, $\langle (bc)(ae) \rangle$, $\langle b \rangle$, $\langle bc \rangle$	$\langle c \rangle$, $\langle ca \rangle:2$, $\langle cb \rangle:3$, $\langle cc \rangle:3$
$\langle d \rangle$	$\langle (cf) \rangle$, $\langle (c(bc)(ae) \rangle$, $\langle (_f)cb \rangle$	$\langle d \rangle$, $\langle db \rangle:2$, $\langle dc \rangle:3$, $\langle dcb \rangle:2$
$\langle e \rangle$	$\langle (_f)(ab)(df)cb \rangle$, $\langle (af)cbc \rangle$	$\langle e \rangle$, $\langle ea \rangle:2$, $\langle eab \rangle:2$, $\langle eac \rangle:2$, $\langle eacb \rangle:2$, $\langle eb \rangle:2$, $\langle ebc \rangle:2$, $\langle ec \rangle:2$, $\langle ecb \rangle:2$, $\langle ef \rangle:2$, $\langle efb \rangle:2$, $\langle efc \rangle:2$, $\langle efc b \rangle:2$
$\langle f \rangle$	$\langle (ab)(df)cb \rangle$, $\langle cbc \rangle$	$\langle f \rangle$, $\langle fb \rangle:2$, $\langle fbc \rangle:2$, $\langle fc \rangle:2$, $\langle fcb \rangle:2$

To reduce the redundant checking at every possible position of potential sequences (first pitfall of FreeSpan), the items within each itemset need to be ordered. Without loss of generality, one can assume that they are always listed

alphabetically. For example, the sequence is written as $\langle c(abd)ab(de) \rangle$ instead of $\langle c(bda)ab(ed) \rangle$.

To reduce the size of projected databases (second pitfall of FreeSpan), one follows the order of the prefix of a sequence and only their suffixes are projected into projected databases instead of considering all possible occurrences of frequent subsequences. In each projected database, sequential patterns are grown by exploring only local frequent patterns. With PrefixSpan, no candidate sequence needs to be generated as PrefixSpan only grows longer sequential patterns from the shorter frequent ones while GSP generates and tests a significant number of candidate sequences.

Moreover, projected databases in PrefixSpan keep shrinking as only the postfix subsequences of a frequent prefix are projected. FreeSpan does not just take postfixes (third pitfall of FreeSpan), but PrefixSpan does.

To summarize, PrefixSpan is more performance preferable than FreeSpan in terms of the following:

1. The redundant checking at every possible position of potential sequences was reduced by assuming that items within each element are always listed alphabetically. Thus, the first cost of FreeSpan is solved.
2. Projected databases keep shrinking since only the suffix subsequences are projected. Usually only small set of sequential patterns grow quite long in a sequence, thus projection in PrefixSpan only takes suffix subsequences. Thus, the second and third costs of FreeSpan are solved.
3. Compared to FreeSpan which needs three time database scans, PrefixSpan scans database only once. The subsequent process counts on the projected databases created in a file.

However, the fourth drawback of FreeSpan, which is the recursive construction of projected databases, is still unable to be resolved by Prefixspan. In the worst case, PrefixSpan builds one projected database for each pattern.

2.2.3.3 PrefixSpan with Pseudoprojection Technique

Pseudoprojection technique was proposed to reduce the number and size of projected databases significantly [11], [12] on the same journal. Unlike projected database, it avoids physically copying suffixes. It uses pointers referring to the sequences in the database. The very basic concept of the pointer-based traversal Pseudoprojection technique is explained as follow.

Instead of generating extravagant projected databases by copying the whole suffixes, one may just register all the position index of the associated customer into an **index set** by means of a (**pointer, position**) pair, where *pointer* is a pointer to the corresponding sequence and *position* represent the position indices of the projected suffix in the sequence. Offset should be an integer, if there is a single projection point; and a set of integers, if there are multiple projection points. Each offset indicates the starting projection position in a sequence.

Based on the concepts, pseudoprojection follows lemmas as shown below.

Lemma 1: Let α be the current pattern, β is the candidate item, and a new candidate pattern γ is yielded by appending β to α . The index set of γ contains all position indices of β that is

- (1) Not located in the same itemset with, and
- (2) Not less than
the smallest index of α .

Proof. If γ is the pattern resulted by appending β to α then there exists α that is not located in the same itemset with β but located before β . If there exists position index of β after position index of α in the different itemset then there exists position index of β after the smallest index of α in the different itemset. Hence, the index set of γ contains all position indices of the index set that is not located in the same itemset with the smallest index of α . If μ is appended to θ then the μ must be located after θ . Thus, the indices of γ must be greater than the index of α . Therefore, we have the lemma. **Q. E. D.**

Lemma 2: Let α be the current pattern, β is the candidate item, and a new candidate pattern γ is yielded by assembling β to α . The index set of γ contains position indices of β located in the same itemset with all indices of α .

Proof. If γ is the pattern resulted by assembling β to α then there exists α that is located in the same itemset with β but located before α due to alphabetical order of items in an itemset. If there exists position index of β after one position index of α in the same itemset then this index of β cannot be applied to the other indices of α . Hence, the index set of γ contains position indices of the index set that is located in the same itemset with all indices of α . Therefore, we have the lemma. **Q. E. D.**

Lemma 3: Let α be the current pattern, β is the candidate item, and a new candidate pattern γ is yielded by either assembling or appending β to α . It still requires checking the in-memory sequence database to build the index set of γ .

Proof. To check whether α is located in the same itemset with β or not, one needs to check the in-memory sequence database. Based on Lemma 1, to build the index set of γ yielded by appending β to α , it is required to check whether all position indices of β are located in the different itemset with the smallest index of α . Based on Lemma 2, to build the index set of γ

yielded by assembling β to α , it is required to check whether position indices of β are located in the same itemset with all indices of α . So, it still requires checking the in-memory sequence database to build the index set of γ . Therefore, we have the lemma. **Q. E. D.**

Below is the example of mining sequential patterns using PrefixSpan with pseudoprojection technique.

Example 2.3: Given the sequence database in Table II with minimum support 50%, Table IV shows the index set for 1-patterns, where \$ indicates that the prefix contributes the occurrence frequency of the sequence but its projected suffix is empty, and \emptyset indicates no occurrence of the prefix in the corresponding sequence.

Table IV. Index set for 1-patterns

Sequence ID	<a>		<c>	<d>	<e>	<f>
1	1,3,7	4	5,8,12	10	\emptyset	\$
2	1,9	6	4,7	2	\$	\emptyset
3	4	5,12	10	7	1	2,8
4	5	10	8,12	\emptyset	1	6

To obtain 2-patterns, reuse the index set for 1-patterns. Suppose that <a> is considered as the current pattern and the possible candidate items are all 1-patterns. Consequently, the candidate patterns are <aa>, <ab>, <(ab)>, <ac>, <(ac)>, <ad>, <(ad)>, <ae>, <(ae)>, <af>, and <(af)>.

The position indices set of <aa> are obtained from current indices of <a> with omitting its first position index for each sequence (see Lemma 1). Thus, pointer 3 and 4 does not support the candidate pattern <aa>, and since the support is no less than the minimum support, <aa> is decided to be pattern.

The position indices set of $\langle ab \rangle$ contains all position indices of $\langle b \rangle$ located at the different itemset with the first index of $\langle a \rangle$ (see Lemma 1). Thus, all pointers support the candidate pattern $\langle ab \rangle$ and this candidate is put as the pattern.

The position indices set of $\langle (ab) \rangle$ contains position indices of $\langle b \rangle$ located in the same itemset with all indices of $\langle a \rangle$ (see Lemma 2). Thus, pointer 2 and 4 do not support the candidate pattern $\langle (ab) \rangle$, and since the support is no less than the minimum support, $\langle (ab) \rangle$ is decided to be the pattern.

The position indices set of $\langle (ac) \rangle$ contains position indices of $\langle c \rangle$ located in the same itemset with all indices of $\langle a \rangle$ (see Lemma 2). Thus, only pointer 1 supports the candidate pattern $\langle (ac) \rangle$, and since the support is less than the minimum support, $\langle (ac) \rangle$ is not considered to be pattern.

The index set of longer candidate pattern $\langle (ab)c \rangle$ contains all position indices of $\langle c \rangle$ located at the different itemset with the first index of $\langle (ab) \rangle$ (see Lemma 1). Thus, pointer 2 and 4 does not support the candidate pattern $\langle (ab)c \rangle$, and since the support is no less than the minimum support, $\langle (ab)c \rangle$ is decided to be pattern. Table V below illustrates position indices for some patterns.

Table V: Index Set for n-patterns ($n > 1$)

Pointer	$\langle aa \rangle$	$\langle ab \rangle$	$\langle (ab) \rangle$	$\langle ac \rangle$	$\langle (ac) \rangle$...	$\langle (ab)c \rangle$...
1	3,7	4	4	5,8,12	5,8	...	8,12	...
2	9	6	\emptyset	4,7	\emptyset	...	\emptyset	...
3	\emptyset	12	5	10	\emptyset	...	10	...
4	\emptyset	10	\emptyset	8	\emptyset	...	\emptyset	...

Pseudoprojection reduces the cost of projection substantially when the database can fit into main memory [4], [11], [12]. The use of physical projections may cause repeatedly copying different suffixes of the sequence,

which is costly. Index position pointers of a sequence may prevent the cost of physical projection of the suffix and, thus, may save both space and time from generating the physical intermediate projected databases. However, it may not be efficient if the Pseudoprojection is used for disk-based accessing since I/O access is costly. Thus, PrefixSpan applies the physical projection if the projected databases cannot be fit into main memory.

Based on the performance study of PrefixSpan [12], FreeSpan ran faster than GSP, SPADE outperformed FreeSpan, and PrefixSpan is the fastest among all those three tested algorithms. PrefixSpan with pseudoprojection is consistently faster than PrefixSpan without pseudoprojection although the gap is not big in most cases. In addition, PrefixSpan consumes almost one order of magnitude less memory than SPADE as well as GSP. This means that PrefixSpan is not only efficient, but also stable in memory utilization, compared to SPADE and GSP. PrefixSpan only needs memory space to hold the sequence data sets plus a set of header tables and pseudoprojection tables. Figure 2 illustrates mining sequential patterns using PrefixSpan with pseudoprojection.

2.2.3.4 MEMISP

In 2005, Ming-Yen Lin and Suh-Yin Lee proposed MEMISP algorithm [13]. A careful analysis of MEMISP reveals that MEMISP seems quite similar to PrefixSpan with pseudoprojection. PrefixSpan with pseudoprojection differs with MEMISP in two cases: (1) when the database can be held in main memory; and (2) when the database cannot be held in main memory.

When the database can be held in main memory, PrefixSpan with pseudoprojection removes infrequent items while MEMISP removes no items. Thus, MEMISP is not efficient since it requires more memory to store the

useless infrequent items and it pointlessly requires time to check for infrequent items in the recursive deeper mining phase.

When the database cannot be held in main memory, PrefixSpan with pseudoprojection generates and scans the projection databases that might be several times the original database size, while MEMISP only scans the database twice or more without generating any projected databases using partition-and-validation technique, which is the only breakthrough in MEMISP. This technique partitions the database, therefore each partition can be held into main memory. The set of potential patterns is obtained by collecting the discovered patterns from each partition. Then the true patterns can be validated by one more database scan for support counting.

Based on the experiments of MEMISP, although MEMISP outperformed GSP, MEMISP has never been compared to the real PrefixSpan with pseudoprojection, which dynamically swaps into PrefixSpan without pseudoprojection when the database cannot fit into main memory. Once PrefixSpan is swapped into PrefixSpan without pseudoprojection, it applies physical projection. Rather, MEMISP was compared to PrefixSpan-1, which applies no pseudoprojection when the database either can or cannot fit into memory, and PrefixSpan-2, which applies pseudoprojection when the database either can or cannot fit into memory. It is reported that PrefixSpan-2 outperforms MEMISP but when the database cannot be held into main memory, PrefixSpan-2 is slowing down due to disk-based pseudo projection.

2.2.3.5 Discussion

Initially Han et al. proposed FreeSpan [10], which outperforms an algorithm based on apriori approach, GSP. However, SPADE [8], which was proposed in the following year, has been proven to run faster than FreeSpan [10].

Subsequently, Pei et al. proposed PrefixSpan and pseudoprojection technique [4], [11], which outperforms SPADE, GSP, as well as FreeSpan [12]. In 2005, Lin and Lee proposed MEMISP [13], which is precisely equal to PrefixSpan with pseudoprojection when the database can fit into main memory, except that MEMISP removes no items. Based on thorough investigation, below are seven reasons why the implementation of PrefixSpan with pseudoprojection outperforms the implementation of other tested algorithms when the database can fit into main memory.

1. PrefixSpan generates no candidates while Apriori approach algorithms must handle the exponential number of candidates. Therefore, Apriori approach takes a great deal of time and substantial amount of memory to generate and test a tremendous number of candidates.
2. PrefixSpan scans database only once, while Apriori approach scans database many times depending on the length of longest pattern, and SPADE as well as FreeSpan scan database three times.
3. Unlike SPADE, PrefixSpan need not transform the database into vertical format.
4. PrefixSpan orders items within each itemset, thus it need not check at every possible position of potential patterns as FreeSpan does.
5. PrefixSpan reduces the size of projected databases by projecting only the suffixes instead of considering all possible occurrences of patterns.
6. The longer the prefix, the more projected databases in PrefixSpan shrink. Moreover, pseudoprojection eliminates the effort of building projected database by only registering the index of suffixes.
7. Pseudoprojection technique prevents repeatedly copying suffixes of each sequence, thus it is efficient in terms of memory space and execution time.

Despite those benefits, PrefixSpan with pseudoprojection bears two major setbacks. First, PrefixSpan with pseudoprojection builds two intermediate databases, i.e. index set (also known as pseudoprojection database) and in-memory sequence database when searching for patterns. If the way PrefixSpan

with pseudoprojection builds and uses those two temporary databases is not managed carefully, then the execution time and memory utilization will be considerably affected. For instance, if the items bought are stored into a dynamic array with a large growth factor, the mining will need bigger memory space to reserve the extension of array that may be unused. If they are stored into a dynamic array with growth factor of 1, the mining will only spend its time to raise the size of array by one. Thus, a framework to store the intermediate databases in PrefixSpan with pseudoprojection is an essential implementation issue to bring up to front.

MEMISP, which is similar to PrefixSpan with pseudoprojection, stores its index set and its in-memory sequence database as a variable-length array in memory [13]. However, using variable-length array to store the whole index set and sequence database into main memory is impossible. The sequence database contains the list of customers and the same customer may come to the shop more than once, thus each customer contains the list of transaction time. Each time customer comes he/she might buy more than one item, thus each transaction time contains the list of items bought. If one uses variable-length array to store just the items bought, then the list of customers and the list of transaction time are missing. Likewise, the index set contains the list of prefixes, each prefix contains the list of pointers, and each pointer contains the list position indices. If one uses variable-length array to store just the position indices of certain customer and for certain prefix, then PrefixSpan misses the list of prefixes and the list of pointers. Thus, to solve this problem, a framework to store the in-memory sequence database and to construct the index set is proposed in the next chapter.

The second setback is the use of in-memory sequence database. Pseudoprojection technique requires to maintain and to visit the in-memory sequence database frequently until all patterns are figured out. Maintaining in-memory sequence database consumes a quite amount of memory space during

mining. Frequent access to in-memory sequence database is not efficient since there are many redundant and unnecessary checks to this copy of original database into memory when the candidate patterns are examined. Earlier elimination of the in-memory sequence database in mining using PrefixSpan with pseudoprojection would certainly reduce the execution time and memory utilization. The proposed work, which will be discussed on the next chapter, indicates that this solution is possible and it is actually improve the performance.

CHAPTER 3

IMPROVEMENTS ON PREFIXSPAN WITH PSEUDOPROJECTION

Chapter 3 describes the two proposed improvements on PrefixSpan algorithm with pseudoprojection for mining sequential patterns. The first proposed improvement technique is SPM-Tree Framework to build the in-memory sequence database and to construct the index set. Using Java as a case study, a series of experiments is conducted to select suitable Collections for the framework. Finally yet importantly, Separator Database is proposed to enable early removal of the in-memory sequence database once the index set for 1-patterns.

3.1 SPM-Tree Framework

3.1.1 The Need for a Framework

PrefixSpan with pseudoprojection builds two intermediate databases, i.e. index set (also known as pseudoprojection database) and in-memory sequence database when searching for patterns. Improper management of building and using those two temporary databases PrefixSpan with pseudoprojection may adversely affect the execution time and memory utilization. For instance, if the items bought are stored into a dynamic array with a large growth factor, the mining will need bigger memory space to reserve the extension of array that may be unused. If they are stored into a dynamic array with growth factor of 1, the mining will only spend its time to raise the size of array by one. Another example, if the items bought are stored into a dynamic array with a very large initial size, the mining will need bigger memory space for this big initial size, which may only be filled with one element. Thus, a framework to store the intermediate databases in PrefixSpan with pseudoprojection is an essential implementation issue to bring up to front.

As previously mentioned, MEMISP stores its index set and its in-memory sequence database as a variable-length array in memory [13] and a plain solution as variable-length array cannot solve problem of storing the index set and sequence database into main memory.

Hence, SPM-Tree is proposed as an efficient framework for index set and in-memory sequence database storage. In a nutshell, SPM-Tree Framework is a tree framework with the following characteristics: (1) all leaves must be located at the same depth that is the max-depth; and (2) the tree size can be two or more. For now, this framework is only used for the purpose of mining sequential patterns using PrefixSpan with pseudoprojection. Further study may figure out that this framework also works for other cases than this algorithm.

3.1.2 Definition of Terms

To illustrate the framework, several definitions need to be introduced.

Definition 5 (Array of Items Bought, Array of Transactions, Array of Customers). Array of items bought is a variable length array containing the list of all items by a customer on a transaction time, denoted as $T_{c,t} = \{i_1, i_2, \dots, i_n\}$, where c is the customer id, t indicates t^{th} time this customer comes, and n is the number of items bought on the t^{th} time customer c comes. Array of transactions is a variable length array containing the list of transactions of customer c , denoted as $C_c = \{T_{c,1}, T_{c,2}, \dots, T_{c,m}\}$, where m is the number of transactions by customer c . Array of customers is a variable length array containing the list of all customers, denoted as $A = \{C_1, C_2, \dots, C_p\}$, where p is the number of customers.

Definition 6: (Array of Offsets, Array of Pointers, Array of Prefixes)

Array of offsets is a variable length array containing the list of all offsets by a prefix on a pointer, denoted as $R_{x,r} = \{o_1, o_2, \dots, o_n\}$, where x is the prefix, r indicates the pointer, and n is the number of offsets for prefix x of pointer r . Array of pointers is a variable length array containing the list of pointers of prefix x , denoted as $X_x = \{R_{x,1}, R_{x,2}, \dots, R_{x,m}\}$, where m is the number of pointers by prefix x . Array of prefixes is a variable length array containing the list of all prefixes, denoted as $B = \{X_1, X_2, \dots, X_p\}$, where p is the number of distinct items.

3.1.3 The SPM-Tree Framework

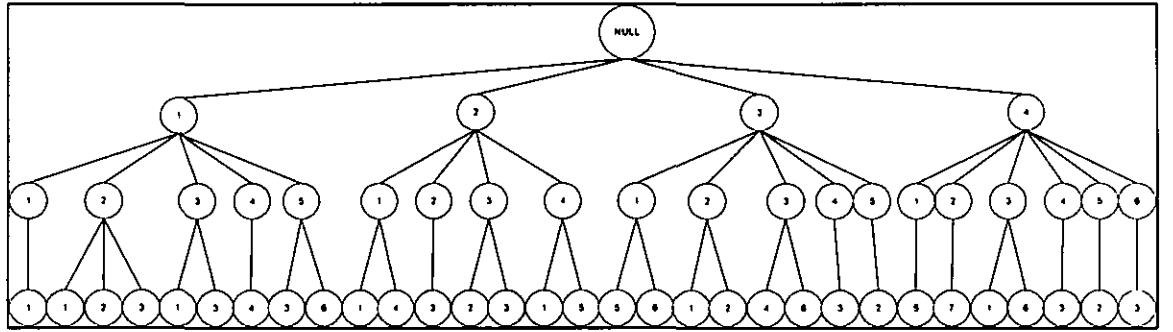


Figure 3. SPM-Tree Framework

Figure 3 above depicts the representation of SPM Tree Framework. The framework is applied to store the sequence database into memory and to construct the index set. The following lemma regarding to the SPM-Tree can be stated.

Lemma 4: Three-level SPM-Tree can be used as framework to store in-memory sequence database and the index set in PrefixSpan with pseudoprojection.

Proof. For in-memory sequence database, let $C = \{C_1, C_2, \dots, C_n\}$ be an array of customers, where n is the number of distinct customers in the sequence

database. C_f is an array of transaction time for customer f , where $C_f = \{C_{f,1}, C_{f,2}, \dots, C_{f,m}\}$, m denotes how many transactions C_f makes, and $1 < f < n$. Then, C_{fj} is an array of items bought by customer f in transaction time j , where $C_{fj} = \{B_1, B_2, \dots, B_k\}$, k denotes how many items bought for transaction time C_{fj} , $1 < j < m$, $B_1 \subseteq I, B_2 \subseteq I, \dots, B_k \subseteq I, I = \{i_1, i_2, \dots, i_p\}$ is the list of distinct items, and p is the number of distinct items. Then the SPM-Tree Framework for in-memory sequence database is C .

For constructing index set, let $A = \{A_1, A_2, \dots, A_p\}$ be an array of prefixes, where p is the number of distinct items. A_x is an array of pointers for item i_x , $1 \leq x \leq p$. $A_x = \{A_{x,1}, A_{x,2}, \dots, A_{x,n}\}$, where n is the number of distinct customers in the sequence database. A_{xf} is the list of offset for index set A_x for customer f , where $1 < f < n$, and $A_{xf} = \{D_1, D_2, \dots, D_y\}$, where y is the occurrence frequency for item i_x on array of customer C_f , $1 \leq y \leq D_y$, $1 < y < \text{length}(C_f)$, where $\text{length}(C_f)$ is the number of items bought by customer C_f + number of transaction times by customer C_f , and $1 < D_1 < \text{length}(C_f)$, $1 < D_2 < \text{length}(C_f)$, \dots , $1 < D_y < \text{length}(C_f)$. Then the SPM-Tree Framework for index set is A . **Q. E. D.**

Based on above lemma, both in-memory sequence database and index set may be stored in three-level SPM-Tree. The following lemma reducing the level of SPM-tree for storing in-memory sequence database can be stated.

Lemma 5: For in-memory sequence database, SPM-Tree can be reduced into two-level-tree by using separator signs to separate different transactions in a sequence. However, this does not prevail for index set.

Proof. Once the in-memory sequence database is created, PrefixSpan with pseudoprojection needs to walk through all sequences on the in-memory sequence database to check whether the index of current pattern is located in the same itemset with the candidate item (Lemma 3). Hence, there is no need to traverse into specific transaction of a sequence just for checking

whether the index of current pattern is located in the same itemset with the candidate item. PrefixSpan with pseudoprojection just needs to know whether there exists an itemset separator between the index of the current pattern and the index of candidate item. Thus, the items bought by a customer could be represented as a list with separator sign to notify the end of current itemset. Each time one goes to index the item of next itemset, one may put a separator sign.

For index set, it is impossible to reduce the three-level framework to become two-level since the retrieval of the offset for certain pointer is still required for support counting. **Q. E. D.**

Below is the example for the application of Lemma 5.

Example 3.1: Let a sequence $\langle (ad)(bc)(ae) \rangle$ is stored in the in-memory sequence database. It is stored on an array of items bought as $[a, d, -, b, c, -, a, e, -]$ without creating any array of transactions.

3.1.4 Data Structure Requirements for The Framework

The framework must be able to run fast for the operations required. Consequently, information about operations required to be executed repetitively by the data structure is necessarily considered. Here are analyses of data structure requirement based on the workflow of PrefixSpan with Pseudoprojection:

1. Constructing the in-memory sequence database

All items bought should be appended to the array of items bought and all arrays of items bought should be appended to the array of customers, which contains arrays of items bought. Thus, appending integers, which are items bought, and appending Objects, which are arrays of items bought, are required to be optimized.

2. Constructing index set for 1-patterns

Constructing index set for 1-patterns requires retrieval on items bought for all customers in the in-memory database and appending their position indices into the index set. Thus, the requirements of data structure on this process are retrieving integers (items bought), retrieving `Objects` (arrays of items bought), appending the integers (offsets), and appending the `Objects` (arrays of offsets and arrays of pointers).

3. Discovering support count

In this stage, the offset for each pointers of certain prefix is exploited. Thus, the requirements of data structure are retrieving integers (offsets) and `Objects` (arrays of offsets and arrays of pointers).

4. Creating the intermediate index set

This stage retrieves the given intermediate index set, i.e. index set of the current pattern, and the earlier-built index set, i.e. index set of the item to be appended or assembled with current pattern. In addition, the projected database yielded needs to be stored by means of whom the support of the new pattern is counted. Thus, the requirements of data structure are retrieving and appending integers (offset) as well as retrieving and appending `Objects` (arrays of offset and arrays of pointers).

To sum up, the SPM-Tree only requires the data structure to be able to append and retrieve integers and `Objects` frequently. It is not necessary to check the other functionalities of the data structure, such as deleting and sorting elements, since it is hardly ever been used in PrefixSpan.

3.1.5 Data Structure Candidates for The Framework

Collections may be useful to store variable-length array. Collections are Objects that groups many elements into a single unit. Collections are used to store, retrieve, manipulate, and transmit data from one type to another. The right choice of Collections could improve the performance. Collections in Java consist of Set, Map, and List [19]. The implementation of Set is slower than the other Collections, and must be avoided except when the functionality of Set is really needed. Meanwhile, a Map is an object that maps keys to values, and cannot contain duplicate keys. Thus, Map cannot be applied to this sort of collection as well. Thus, the only possible Collection to apply is List. A List is an ordered Collection (sometimes called a sequence) that may contain duplicate elements. In addition, List includes operations for positional access, search, list iteration, and range-view. List has a dynamic array based implementation, facilitating the fast-indexing construction, and able to search very fast when searching on an index position. Java Development Kit has four general-purpose List implementations, i.e. Vector, Stack, ArrayList, and LinkedList. LinkedList implements a doubly linked list, Vector is a synchronized List, and Stack is subclass of Vector, which has the same performance characteristic but has an additional functionality, which is LIFO. Meanwhile, ArrayList is an asynchronous List.

Java Development Kit (JDK)'s Collections are not type-specific, thus programmers must cast the data type each time the elements are being used. Jakarta Commons Primitives provides primitive Collections. Moreover, these primitive Collections run faster than their java.util equivalents. A primitive collection of integer named ArrayIntList saves 75% of memory space [39].

Therefore, the data structure candidates for storing integer data and storing arrays are `ArrayList`, `Vector`, `Stack`, and `LinkedList`. For storing integer data, `ArrayIntList` is an additional collection to try since `ArrayIntList` can only keep integers. The results of a series of experiments to select the suitable candidates for SPM-Tree Framework are later described in Chapter 4.

3.2 Separator Database

3.2.1 Background

Let us consider `PrefixSpan` algorithm.

Algorithm 1 (*PrefixSpan*)

Input : Sequence database S , minimum support $min_support$

Output : A complete set of sequential patterns

Method :

1) *PrefixSpan*($\langle \rangle, 0, S$)

procedure *PrefixSpan* ($\alpha, L, S|_{\alpha}$)

The parameters are 1) α is a sequential pattern; 2) L is the length of α ; and 3) $S|_{\alpha}$ is the α -projected database if $\alpha \neq \langle \rangle$, otherwise, it is the sequence database S .

- 1) Scan $S|_{\alpha}$ once, find each frequent item, b , such that :
 - a) b can be assembled to the last element of α to form a new sequential pattern.
 - b) $\langle b \rangle$ can be appended to α to form a new sequential pattern.
- 2) For each frequent item b , append it to α to form a sequential pattern α' and output α' .
- 3) For each α' , construct α' -projected database $S|_{\alpha'}$.
- 4) *PrefixSpan* ($\alpha', L+1, S|_{\alpha'}$)

It is costly to keep the in-memory sequence database until mining is over. One must pay for more space to store the in-memory sequence database and for redundant checking on every single original sequence at deeper mining phase. As per Lemma 3, PrefixSpan with pseudoprojection requires the in-memory sequence database in deeper mining phase to build the index set of all patterns except 1-patterns. A new technique is needed to replace the functionality of the in-memory sequence database. Hence, Separator Database is proposed to cope with this necessity. Unfortunately, in-memory sequence database is required to build the index set of frequent items as well as to construct the Separator Database. To build the Separator Database, PrefixSpan with pseudoprojection requires the original database to record all offsets of every separator signs for each sequence, thus in the extraction phase the in-memory sequence database cannot be removed. To build the index set of frequent items, PrefixSpan with pseudoprojection requires the original database to record all offsets of every certain frequent item for each sequence, thus in the initial mining phase, the in-memory sequence database cannot be removed. The in-memory sequence database can only be removed in deeper mining phase, since it is merely used to check whether there are any separator signs between the index set of current patterns and the index set of candidate items.

3.2.2 Definition of Terms

The concepts of separator list, Separator Database, and separator location need to be introduced at first.

Definition 7 (Separator List, Separator Database, Separator Location)

Separator list is the list of separator sign indices on each sequence. Separator Database is the collection of separator list for all sequences. Let $\eta = \{i_1, i_2, \dots, i_{|\eta|}\}$ be the index set of the current pattern of certain sequence and $\kappa = \{s_1, s_2, \dots, s_{|\kappa|}\}$ be the separator list of the associated

sequence. Separator location of i_μ (for $1 < \mu < |\eta|$) is denoted as $\left\{ r \mid r \in \kappa \wedge r = \min_r(r > \mu) \right\}$.

3.2.3 The Separator Database

The Separator Database is created when the in-memory sequence database is scanned at the early stage of PrefixSpan. After the construction of index set for all frequent items, the database can be deleted from memory. The mining process may proceed without in-memory sequence database. The following lemma states the correctness of replacing in-memory sequence database to build the index set of frequent k-patterns.

Lemma 6: Let α be the current pattern (α is not empty), β is the candidate item, and a new candidate pattern γ is obtained by either assembling or appending β to α . Separator Database can replace the in-memory sequence database to build the index set of γ .

Proof. To check whether α is located in the same itemset with β or not, one need to check the in-memory sequence database. Based on Lemma 1, to build the index set of γ constructed by appending β to α , it is required to check whether all position indices of β are located in the different itemset with the smallest index of α . The Separator Database contains the collections of the list of separator sign indices. To check whether all position indices of β are located in the different itemset with the smallest index of α , one just needs to compare the index of α and the closer but higher index of separator than α , then all position indices of β after this separator location is obtained. Based on Lemma 2, to build the index set of γ constructed by assembling β to α , it is required to check whether position indices of β are located in the same itemset with all indices of α . To check whether position indices of β are located in the same itemset with all indices of α , one just needs to compare the index of α and the closer but

higher index of separator than α , then all position indices of β before this separator location and after α is obtained. Thus, the role of in-memory sequence database can be replaced with Separator Database. Therefore, we have the lemma. **Q. E. D.**

The construction of index set for longer pattern proceeds as following conditions.

Condition 1: If the candidate item is assembled to the last element of current pattern then the index set of candidate pattern is the index of candidate item after current pattern but before the current pattern's separator location.

Condition 2: If the candidate item is appended to the current pattern then the index set of candidate pattern is the index of candidate item after the current pattern's separator location.

Let us examine how to mine sequential patterns using the Separator Database based on the running example.

Example 3.2: Given the sequence database in Table II with minimum support 50%, the Separator Database is as shown in Table VI.

Table VI. Separator Database

Sequence ID	Separator List
1	2,6,9,11,14
2	3,5,8,11
3	3,6,9,11,13
4	2,4,7,9,11,13

To obtain 2-patterns index set, the index set for 1-patterns is employed, as shown in Table IV, and the Separator Database. Suppose that $\langle a \rangle$ is

considered as the current pattern and the possible candidate items are all 1-patterns. Consequently, the candidate patterns are $\langle aa \rangle$, $\langle ab \rangle$, $\langle (ab) \rangle$, $\langle ac \rangle$, $\langle (ac) \rangle$, $\langle ad \rangle$, $\langle (ad) \rangle$, $\langle ae \rangle$, $\langle (ae) \rangle$, $\langle af \rangle$, and $\langle (af) \rangle$.

To get indices of pattern $\langle ac \rangle$ of sequence id 1, one can apply Condition 2. The first index of $\langle a \rangle$ is 1, the separator location is 2, then all indices in $\langle c \rangle$ with indices number closer but higher than 2 are $\{5, 8, 12\}$. Thus, the index set is obtained. However, the index list of $\langle e \rangle$ of sequence id 1 is empty, thus there are no indices for candidate pattern $\langle ae \rangle$. The index list of $\langle ac \rangle$ and $\langle ae \rangle$ for other sequences, as well as the index set of $\langle ab \rangle$, $\langle ad \rangle$, and $\langle af \rangle$ are generated similarly.

To get indices of candidate pattern $\langle (ab) \rangle$ of sequence id 1, one can apply Condition 1. The first index of $\langle a \rangle$ is 1, the separator location is 2 but there are no index of $\langle b \rangle$ between 1 and 2. Pull up to the next index of $\langle a \rangle$, which is 3, the separator location is 6, and index of $\langle b \rangle$ is found between 3 and 6, that is 4. Move to the next index of $\langle a \rangle$, which is 7, the separator location is 9, but again there are no index of $\langle b \rangle$ between 7 and 9. Since 7 is the last element of the index list of $\langle a \rangle$, the index set for $\langle (ab) \rangle$ is $\{4\}$.

Likewise, to get indices of candidate pattern $\langle (ac) \rangle$ of sequence id 1, one still can apply Condition 1. The first index of $\langle a \rangle$ is 1, the separator location is 2, but there are no index of $\langle c \rangle$ between 1 and 2. Pull up to the next index of $\langle a \rangle$, which is 3, the separator location is 6, and index of $\langle c \rangle$ is found between 3 and 6, which is 5. Move to the next index of $\langle a \rangle$, which is 7, the separator location is 9, and index of $\langle c \rangle$ is found between 7 and 9, that is 8. Thus, the index set for $\langle (ac) \rangle$ is $\{5, 8\}$. The index list of $\langle (ab) \rangle$, $\langle (ac) \rangle$ for other sequences, as well as the index set of $\langle ad \rangle$, $\langle ae \rangle$, and $\langle af \rangle$ are generated similarly.

Similarly, to obtain indices of candidate pattern of longer patterns, for instance $\langle(ab)c\rangle$ of sequence id 1, one can apply Condition 2. The first and only index of $\langle(ab)\rangle$ is 4, the separator location is 6, and all indices in $\langle c\rangle$ with index number closer but higher than 6 are 8 and 12. Thus, the index set of $\langle(ab)c\rangle$ is $\{8, 12\}$.

Finally, the complete set of patterns is found in Table III.

3.2.4 Effects of Separator Database to Several Database Characteristics

Besides examining the impact on memory usage, scalability tests need to be conducted to examine how Separator Database affects numerous sequence database characteristics on CPU performance. The following four sequence database parameters are presumed to affect the CPU performance of Separator Database.

- **User specified minimum support threshold.** There are much more patterns figured out for smaller minimum support threshold. The lower minimum support threshold, the more candidate patterns having smaller support that is greater than or equals to the minimum support threshold. Thus, the lower minimum support threshold, the more candidate patterns is allowed to be the sequential patterns and the more recursions occurred in mining patterns. Besides, minimum support is the classical measurement of performance used in previous research.
- **Dataset density.** Suppose a sequence database having a certain number of customers, average number of transactions per customer, average items bought per transaction, and number of distinct items sold in a retail store. If the number of distinct items sold is multiplied while the other parameters (number customer, average number of transactions per customer, average items bought per transaction) are still the same, then the number of distinct items in the same sequence will increase. In other word, the sequence will contain more various items, while the size of sequence and the size of itemsets stay the same. We can also say that the database

gets denser while we add more distinct items sold. When the number of items escalates, the size and length of Separator Database stay the same, but the redundancy of checking in-memory sequence database is reduced to the length of Separator Database. If the length of Separator Database is twice smaller then the number of checking is reduced to twice and Separator Database will help to mine twice faster.

- **Dataset size.** The number of customer sequence belongs to the size of sequence database. The larger dataset size, the more rows of sequence database is to be processed. Larger dataset size also means larger Separator Database, since the size of a sequence database equals to the size of its associated Separator Database. Thus, the size of dataset size affects the performance and scalability of Separator Database.
- **Average itemset length.** Suppose that a consumption rate of customer increases. Thus, the average number of distinct items bought per transaction is also boosted. It means that the average itemset length rises, while the number of customer, average number of transactions per customer, and number of distinct items sold reach a plateau. Consequently, the length of sequence increases but the number of itemsets is on a steady state. The average of itemset length should be the most influencing factor to the performance and scalability of Separator Database since length of the in-memory sequence database rises but the length of Separator Database is changeless. Thus, there are redundant checks on the in-memory sequence database as the average itemset length grows, while the frequency of checking on Separator Database is in a stable state. Besides, when the average itemset length is constant, the frequency of checking on the Separator Database is less than then one on the in-memory sequence database.

A series of experiments have been conducted to see the effects of these database characteristics on Separator Database. The experiment details and results are described further in Chapter 4.

CHAPTER 4

RESULTS AND DISCUSSION

In this chapter, the performance of the five Java Lists for the proposed SPM-Tree Framework is examined. Subsequently, the performance of PrefixSpan with pseudoprojection technique, with Separator Database versus without Separator Database on a series of small databases is conducted. The detailed experimental result figures for the graphs shown in this chapter are shown on Appendix A.

4.1 Selection of Suitable Data Structure for SPM-Tree Framework

The experiments on this subsection aim to select the suitable Java List for our proposed SPM-Tree Framework. The suitably chosen Java Lists for SPM-Tree Framework should contribute towards efficiency improvement of PrefixSpan with pseudoprojection in terms of speed and memory when it is implemented on Java. Based on preceding chapter, SPM-Tree Framework requires:

1. A List having high capability of frequently appending and retrieving integer, and
2. A List having high capability of frequently appending and retrieving Object.

For this purpose, the experiment scenarios are based on those two requirements. Performance study was conducted on five Java Lists, i.e. ArrayList, LinkedList, Vector, Stack, and ArrayIntList. All experiments were conducted on an Intel ® Pentium 4 CPU 3.20 GHz (2

CPUs) with 64 MB memory, running Microsoft Windows XP Professional (5.1, Build 2600) and implementing Java using JDK 1.6.

4.1.1 CPU Performance of Appending Integers into Different List Collections

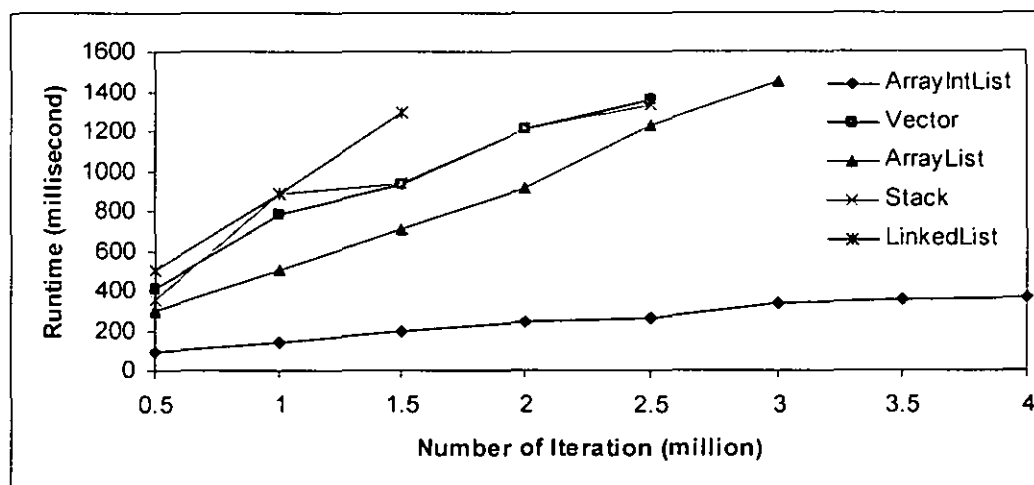


Figure 4. CPU performance of appending integers into different List Collections

Figure 4 shows the processing time of the five Java Lists at different sizes of integer to be appended. Among all five tested Java List collections, LinkedList is the slowest and earliest to give up as the number of element appended escalates. Choosing LinkedList to append any Objects repeatedly is not a good pick due to poor locality and separated memory allocation for each new element appending. LinkedList needs extra storage for references, thus it requires more memory and it cannot append on a relatively small size of array. Meanwhile, Stack and Vector, which apply dynamic array, run better than LinkedList since they do not have any memory allocation for references. The method to append new element in Stack extends the method to append new element in Vector, thus Stack and Vector have a quite similar runtime performance on appending element.

However, `ArrayList`, which grows its capacity into $\frac{1}{2}old_capacity + 1$ each time it is full, outperforms `Vector` and `Stack`, which grow their capacities twice each time they are full. Nevertheless, `ArrayIntList` is the clear winner among all five tested Java Lists for appending either large or small number of integer element. `ArrayIntList` (331.4 seconds) runs over four times more rapidly than `ArrayList` (1,450 seconds) and `ArrayList` needs more than 64 MB memory to finish mining when 3.5 million of integer elements are appended. The main reason is that an array of integer is declared and initialized as the constructor of `ArrayIntList` is invoked, while the constructor of `ArrayList` contains the declaration and initialization of an array of `Object`, which is not a primitive data type.

4.1.2 CPU Performance of Retrieving Integers into Different List Collections

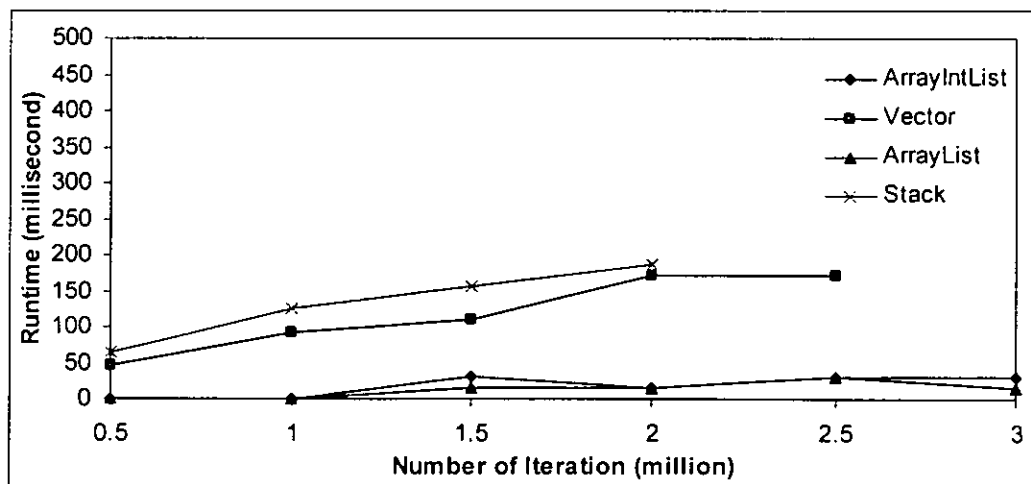


Figure 5. CPU performance of retrieving integers into different List Collections

Figure 5 shows the processing time of element retrieval of the five Java Lists at different sizes of integer set. `LinkedList` is too slow compared

to the rest of four tested Java Lists. Indeed, `LinkedList` is not up to scratch for random access since `LinkedList` does not implement `RandomAccess` interface. Thus, there is no quick way for `LinkedList` to retrieve the k^{th} element other than a sequential scan from the front or back of the `LinkedList`, which is very slow. Since `Stack` and `Vector` apply dynamic array and implement `RandomAccess` interface, they run better than `LinkedList`. `Stack` and `Vector` also run quite similarly because method of accessing an element in `Stack` extends one in `Vector`. However, the performance of random access of `ArrayList` and `ArrayIntList` is much better than `Stack` and `Vector` since `Stack` and `Vector` are synchronized, while `ArrayList` and `ArrayIntList` are not. Synchronized methods are a bit expensive in terms of performance because Java Virtual Machine must lock the object whenever it finds synchronized methods. `ArrayIntList` only performs slightly better than `ArrayList`, yet generally both of them perform comparably similar in terms of speed and both of them are the most scalable among the rest of tested Java Lists.

4.1.3 Memory Usage of Appending Integers into Different List Collections

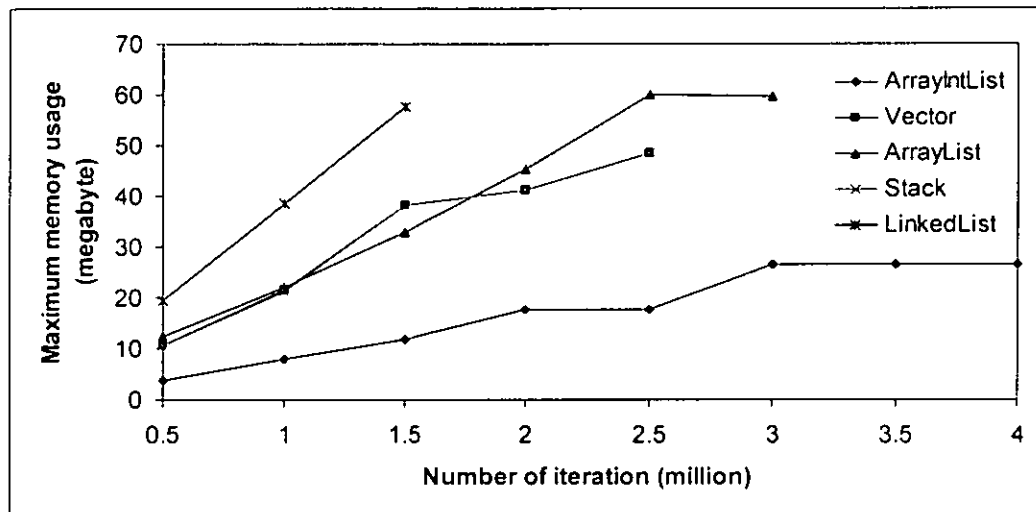


Figure 6. Memory usage of appending integers into different List Collections

Figure 6 shows the memory utilization of the five Java Lists at different sizes of integer to be appended. The performance of `LinkedList` is still the worst, followed by `Stack`, `Vector`, and `ArrayList`. At 500,000 and 1,000,000 integer data, `Stack` (10.65 MB and 21.39 MB) and `Vector` (10.69 MB and 21.24 MB) performs slightly better than `ArrayList` (12.21 MB and 21.98 MB). However, `ArrayList` (32.82 MB) performs slightly better than `Stack` and `Vector` (38.2 MB) when appending 1,500,000 integer data into the Lists. `Stack` and `Vector` again outperform `ArrayList` when appending more than 1,500,000 integer data, and it is getting clearer than before. This fluctuation happens since `Stack` and `Vector` double their capacity, while `ArrayList` raises its capacity into $\frac{3}{2}old_capacity + 1$. The raise of `ArrayList` capacity triggers fluctuation on its runtime. Thus, owing to the fluctuation, `Stack` and `Vector` terminate their executions when 3,000,000 of integer are appended and `ArrayList`

remains continuing. Nevertheless, `ArrayIntList` exploits much less memory than `ArrayList` and still runs when 3,500,000 integer data is appended while `ArrayList` stops running. This shows that casting `Object` into integer for storing integer value saves a lot of memory as well as speeds up the appending and the access.

4.1.4 Memory Usage of Retrieving Integers into Different List Collections

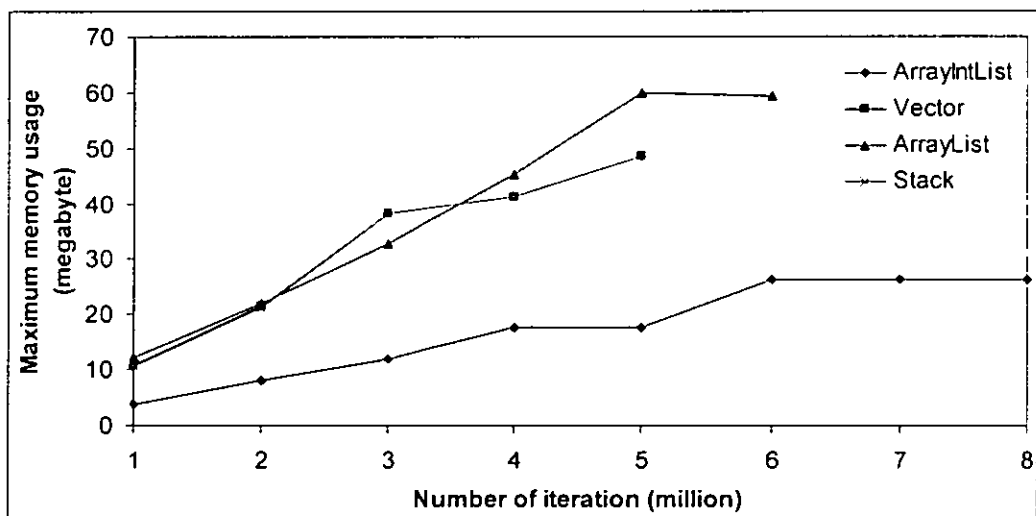


Figure 7. Memory usage of appending and retrieving integers into different List Collections

Figure 7 shows the memory utilization of element retrieval of the five Java Lists at different sizes of integer set. For this test, the memory used for appending elements is counted as well, since it is not plausible to test the memory usage of retrieving of an empty List. The memory utilization of retrieving and appending integer data are quite similar with the one of appending integer data, except that `LinkedList` uses too much memory for both appending and accessing integer data. With the exception of `LinkedList`, this shows that integer retrieval requires almost no memory.

4.1.5 CPU Performance of Appending Objects into Different List Collections

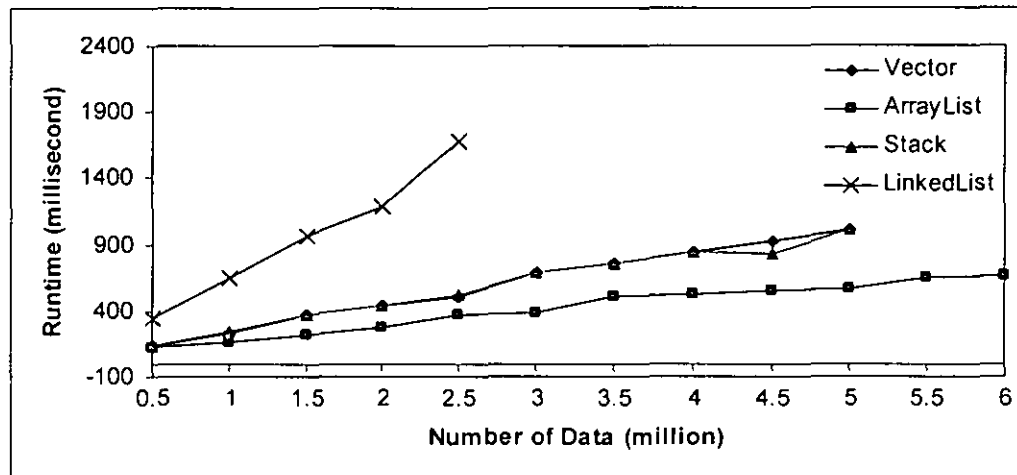


Figure 8. CPU performance of appending Objects into different List Collections

Figure 8 shows the processing time of Stack, Vector, LinkedList, and ArrayList at different sizes of Objects to be appended. Each Object in this test is an array of 30 integer data. Changes in the size of the array does not change the performance since Java considers the array of any size as an Object and even if it does, the size of array is considered as constant variable for comparison purpose. In addition, since ArrayList only stores integers, it is excluded from this test. The graph shows that the performance is roughly similar to the CPU performance of appending integer data. Among all four tested Java List collections, LinkedList is still an unsatisfactory for appending Objects. Stack and Vector run better than LinkedList, and ArrayList outperforms Vector and Stack.

4.1.6 CPU Performance of Retrieving Objects into Different List Collections

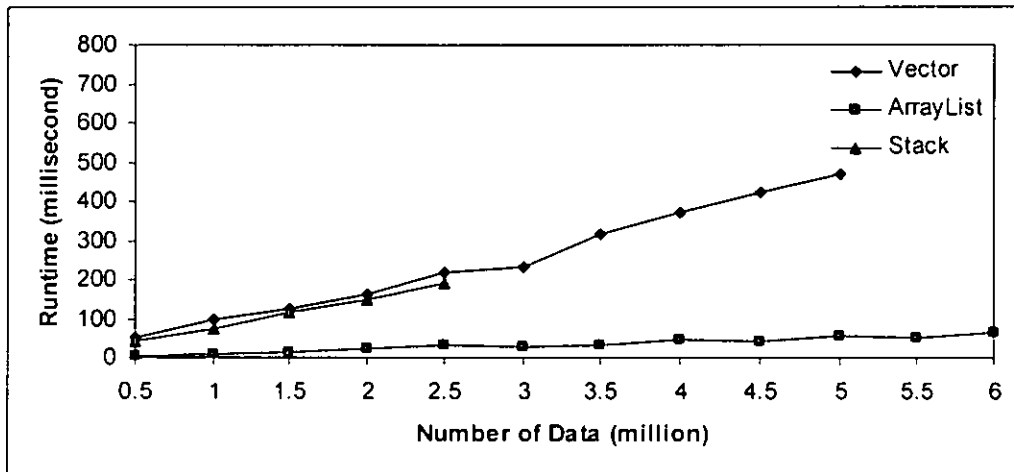


Figure 9. CPU performance of retrieving Objects into different List Collections

Figure 9 shows the processing time of random access among four Java Lists at different sizes of Objects. Each Object in this test is an array of 30 integer data. Again, the graph shows that the performance is roughly similar to the CPU performance of retrieving integer data. Among all four tested Java List collections, LinkedList is still a substandard for storing Objects. Stack and Vector run better than LinkedList, and ArrayList outperforms Vector and Stack.

4.1.7 Discussion

Based on the results of all experiments conducted, LinkedList would never be chosen since the performance is much worse than Stack, Vector, ArrayList, and ArrayIntList, either for storing integer data or for storing Object data. Although the runtime performance of Stack and

Vector are better than LinkedList, they are worse than the runtime performance of ArrayList. Thus, excluding ArrayList, ArrayList is the most suitable choice for storing Object, compared to Stack, Vector, and of course LinkedList.

Whilst Stack, Vector, and ArrayList are used to store Object, ArrayList is specially designed for storing integer primitive data types [20]. Since byte per element of ArrayList is four times smaller than ArrayList of Integer, ArrayList could save 75% of memory space. With this low memory requirement, as our studies results showed, ArrayList outperforms the other four List implementations for storing integer data. Moreover, ArrayList reduces the uncertainty of the type of elements by pre-specifying it as an integer primitive data type, while ArrayList of Integer needs three times more memory space than ArrayList to store an integer. Thus, ArrayList is the most suitable choice for storing integer data, compared to ArrayList, Stack, Vector, and for sure LinkedList.

4.2 Performance and Scalability Test of Separator Database

The subsequent tests compare performance of PrefixSpan with pseudoprojection [11], [12] with Separator Database versus without Separator Database on a series of small databases. The following series of scalability test aim to examine how Separator Database affects the execution time of PrefixSpan with pseudoprojection with regards to user specified minimum support threshold, dataset size, average itemset length, and dataset density. The set of experiments also examine how Separator Database affects the performance of PrefixSpan with pseudoprojection in terms of memory utilization. The synthetic datasets used in our experiments were generated using publicly available Quest Synthetic Data Generation Code for

Association and Sequential Patterns software [5]. The same data generator has been used in most studies on sequential pattern mining [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. The datasets generated mimics real-world transactions. Customers might come once or more, and each time they come they might buy one or more items. The number of transactions per customer and the number of items bought per transaction are set around a mean. All experiments were conducted on an Intel ® Pentium 4 CPU 3.20 GHz (2 CPUs) with 2 GB memory, running Microsoft Windows XP Professional (5.1, Build 2600) and implementing Java using JDK 1.6.

PrefixSpan with pseudoprojection actually has never been implemented in Java. Owing to fair comparison, it is assumed that PrefixSpan with pseudoprojection, either with or without Separator Database, follows the SPM-Tree Framework and it stores arrays of integer and arrays of Objects in `ArrayIntList` and `ArrayList` data structure, respectively.

4.2.1 CPU Performance versus Minimum Support

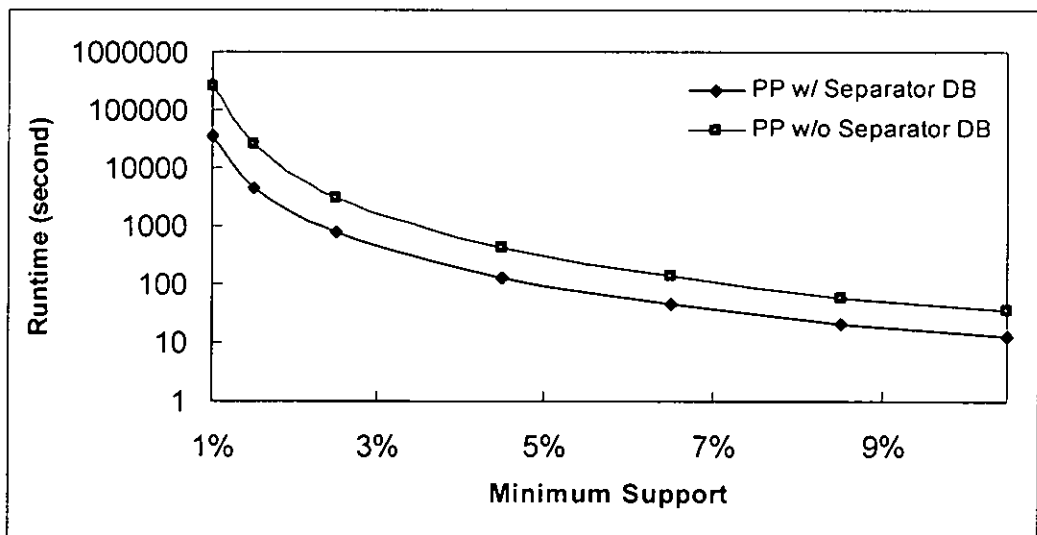


Figure 10. CPU Performance versus Minimum Support on Dataset-1

Figure 10 shows the impact of Separator Database with alteration on minimum support threshold to CPU performance on Dataset-1 (C10k-N50-T2-S10-t4-i2), which contains 10,000 sequences and 50 distinct items. Both the average number of items in a transaction and the average number of transactions in a sequence are set to 2. On average, a frequent sequential pattern consists of four transactions, and each transaction is composed of 2 items. The parameters of Dataset-1 are set to show obviously the impact of Separator Database to the minimum support threshold. When the minimum support is 10%, PrefixSpan with pseudoprojection with Separator Database (runtime = 13.078 seconds) is almost three times faster than PrefixSpan with pseudoprojection without Separator Database (runtime = 36.437 seconds). However, as the minimum support threshold dwindles to 1%, PrefixSpan with pseudoprojection with Separator Database (runtime = 4,410.953 seconds) runs almost one order of magnitude faster than PrefixSpan with pseudoprojection without Separator Database (runtime = 26,276.110 seconds). It is shown that when the minimum support gets lower, Separator Database helps PrefixSpan with pseudoprojection to mine exponentially faster. The main reason is that there are much more patterns found for smaller minimum support threshold.

4.2.2 CPU Performance versus Dataset Density

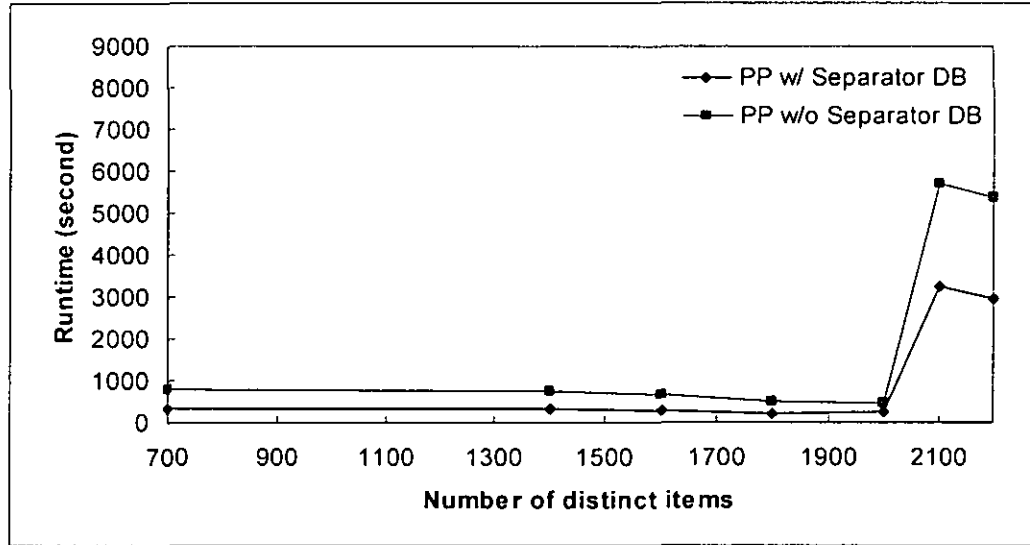


Figure 11. CPU Performance versus Dataset Density on Dataset-2

Figure 11 shows the impact of Separator Database with alteration on the database density to CPU performance on Dataset-2 (C10k-T2-S2-t4-i6), which contains 1000 sequences with the number of items growing from 700 to 2200. Both the average number of items in a transaction and the average number of transactions in a sequence are set to 2. On average, a frequent sequential pattern consists of four transactions, and each transaction is composed of 6 items. The minimum support is set to 1%. The parameters of Dataset-2 are set to show obviously the impact of Separator Database to the dataset density. Consistently, Separator Database improves the performance of PrefixSpan with pseudoprojection to approximately as much as twice faster. It is probably not a big deal when the dataset contains 700 distinct items (runtime with Separator Database = 354.031 seconds, runtime without Separator Database = 809.687 seconds), but it will be a significant disparity when the database contains 2100 distinct items (runtime with Separator Database = 3261.688 seconds, runtime without Separator Database = 5725.344 seconds). The sudden increase may occur anytime due to the steep increase on number of

patterns resulted from the parameter of distribution. The dataset density, in fact, affects the performance of PrefixSpan with pseudoprojection due to the absence of redundant checks to the copy of sequence database into memory. However, the ratio of improvement nearly reaches a plateau since the density of database does not affect the size and the length of both in-memory sequence database and Separator Database.

4.2.3 CPU Performance versus Dataset Size

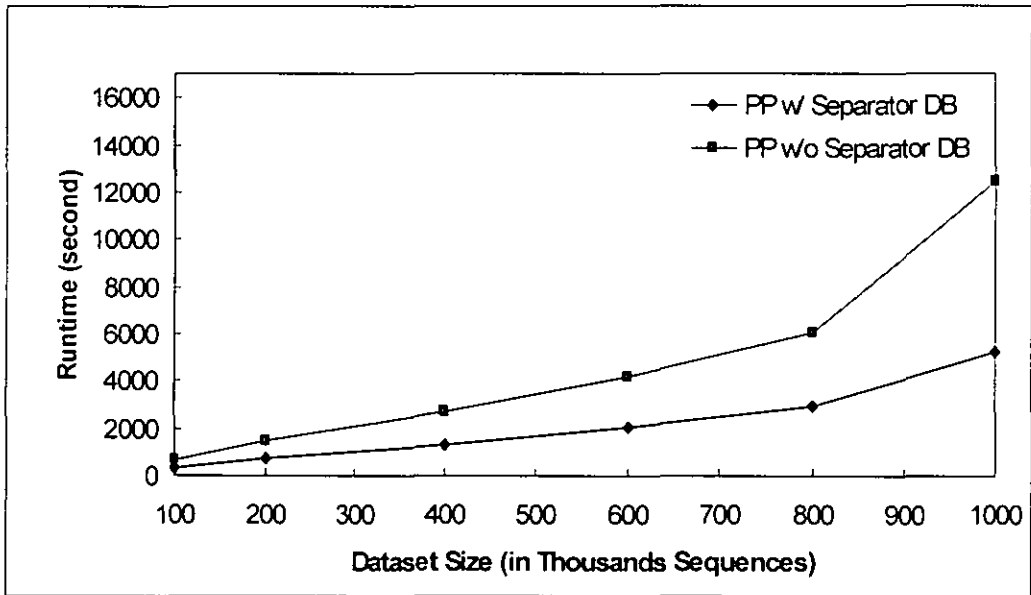


Figure 12. CPU Performance versus Dataset Size on Dataset-3

Figure 12 shows the results of scalability test of PrefixSpan with pseudoprojection, with Separator Database and without Separator Database on Dataset-3 (N10-T2-S2-t4-i2), with the database size growing from 100K to 1,000K sequences with minimum support threshold is set to 1%. The parameters of Dataset-3 are set to show obviously the impact of Separator Database to the dataset size. Separator Database makes the performance of PrefixSpan with pseudoprojection be marginally better as the database size grows. For the database of 100K sequences, PrefixSpan with pseudoprojection

with Separator Database (runtime = 343.969 seconds) is twice faster than PrefixSpan with pseudoprojection without Separator Database (runtime = 695.125 seconds). However, when the database grows to 1,000K sequences, Separator Database (runtime = 5,269.828 seconds) improves more than twice faster than PrefixSpan with pseudoprojection without Separator Database (runtime = 12,503.97 seconds). More patterns to be checked cause the growth of the ratio of improvement.

4.2.4 CPU Performance versus Transaction Length

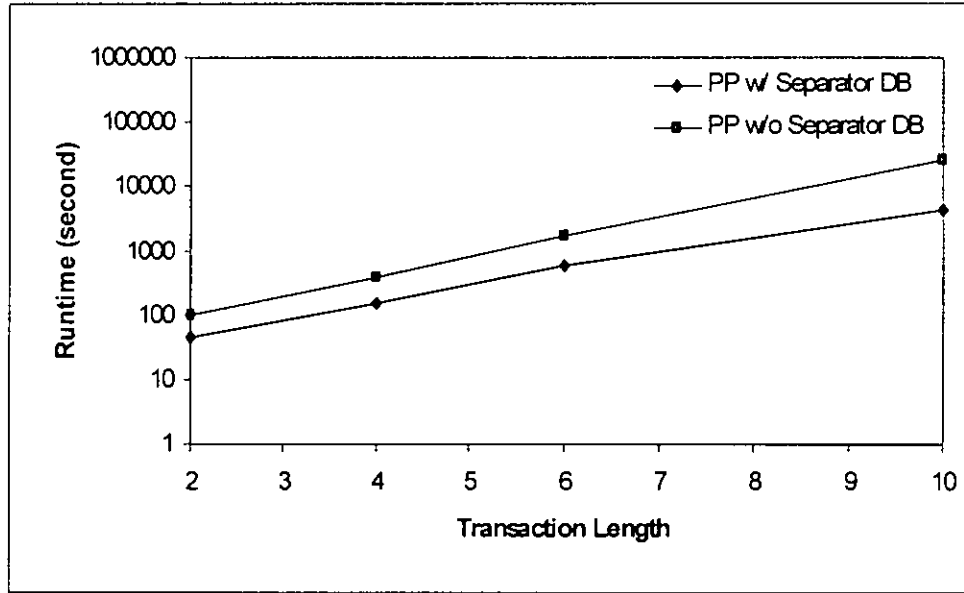


Figure 13. CPU Performance versus Transaction Length on Dataset-4

Figure 13 shows the effect of consumption rate of customer on Dataset-4 (C1k-N50-T2-t4-i2), with the average number of distinct items bought per transaction grows from 2 to 10 with minimum support 1%. Separator Database marginally improves PrefixSpan with pseudoprojection as the customer buy more distinct items in each transaction. For the average distinct items per transaction of 2, PrefixSpan with pseudoprojection with Separator Database (44.86 seconds) runs more than twice as fast as PrefixSpan with pseudoprojection without Separator Database (runtime = 94.25 seconds).

When the average distinct item per transaction grows to 6, PrefixSpan with pseudoprojection with Separator Database (runtime = 566.641 seconds) is nearly three times faster than PrefixSpan with pseudoprojection without Separator Database (runtime = 1,655.671 seconds). Moreover, when the average distinct items per transaction grow to 10, PrefixSpan with pseudoprojection with Separator Database (4,410.953 seconds) is six times faster compared to PrefixSpan with pseudoprojection without Separator Database (26,276.11 seconds). It is plausible since in larger transaction length, checking on in-memory sequence database is much more frequent than checking on Separator Database.

4.2.5 Memory Usage versus Transaction Length

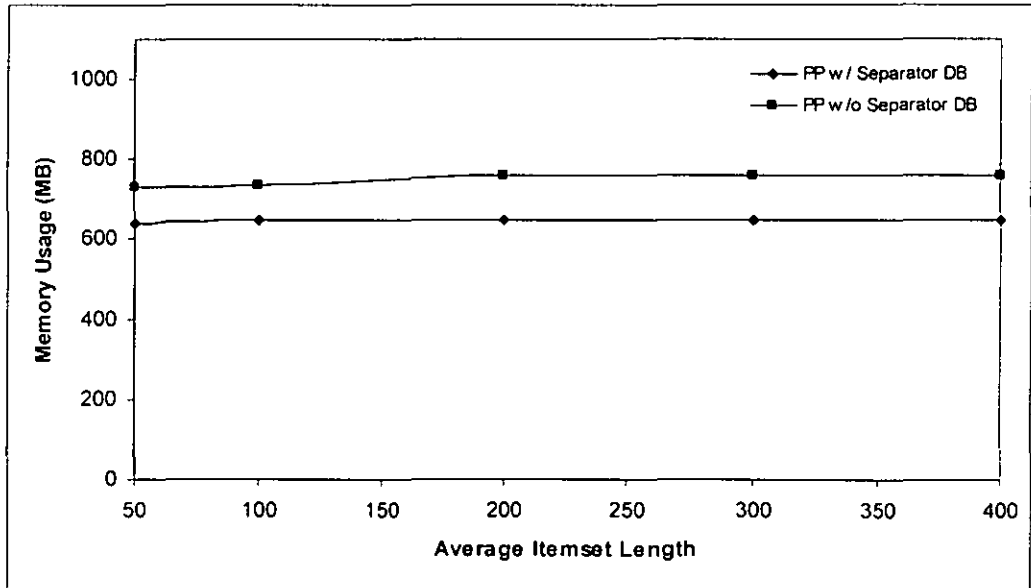


Figure 14. Memory Usage versus Transaction Length on Dataset-4

Finally, the memory usage is compared using Dataset-4 (C1k-N50-T2-t4-i2). Figure 14 shows the effect of consumption rate of customer to the memory usage on Dataset-4, with the average number of distinct items bought per transaction grows from 50 to 400 with minimum support 0.5%, from which it

is clear that Separator Database improve PrefixSpan with pseudoprojection in terms of memory utilization. For the average distinct items per transaction of 50, PrefixSpan with pseudoprojection without Separator Database consumes 732.60 MB while PrefixSpan with pseudoprojection with Separator Database consumes only 637.19 MB, which is 95.42 MB less memory. However, the growth of average itemset length triggers the memory requirement for PrefixSpan without pseudoprojection without Separator Database. For the average distinct items per transaction of 400, PrefixSpan with pseudoprojection without Separator Database consumes 760.52 MB while PrefixSpan with pseudoprojection without Separator Database consumes only 647.37 MB, which is 113.16 MB less memory. It is plausible since storing in-memory sequence database requires more memory than storing Separator Database. The length of Separator Database is always less than the length of its associated in-memory sequence database. Moreover, if the length of itemset is big, then the length of Separator Database will be much smaller than its associated in-memory sequence database.

4.2.6 Discussion

Based on the experiment results from previous subsections, it was shown that Separator Database, as the substitution of in-memory sequence database, improves PrefixSpan with pseudoprojection. With Separator Database, there is no need to traverse along all items inside all data sequences, thus reducing the frequent visiting database when mining and speed up the mining process. The question becomes why Separator Database improve the performance of PrefixSpan with pseudoprojection: Is it because of some implementation tricks, or is it inherently in the algorithm itself? The following analyses explain:

- The in-memory sequence database is removed once index set of 1-patterns are constructed, and Separator Database carries out this in-memory database. Removing in-memory sequence database and replacing it with

Separator Database implies releasing some memory space. Thus, Separator Database saves memory utilization when mining sequential patterns using PrefixSpan with pseudoprojection.

- Separator Database substitutes the complete sequence database in memory only with the list of separator locations for each customer. Using the smaller substitute of in-memory sequence database reduces redundant checking effort when counting the support of a candidate patterns and the algorithm merely checks on the separator location index. Checking on in-memory sequence database takes more processing time than executing basic mathematical comparison operation in Separator Database.

Now let us analyze the results based on four data characteristics. For different minimum support thresholds (Figure 10), it is interesting to note that Separator Database speeds up PrefixSpan with pseudoprojection in exponential ratio. This indicates that the deletion of in-memory sequence database and the use of Separator Database could improve PrefixSpan with pseudoprojection as the minimum support gets lower.

As can be observed in Figure 11, Separator Database roughly speeds up twice as fast as PrefixSpan with pseudoprojection due to the shrinkage of in-memory sequence database into Separator Database. Dataset density merely varies the items distribution in the dataset but it does not change the transaction length, while Separator Database will show its strength when the transaction size is long. As the dataset gets denser, the size of Separator Database is proportional to the in-memory sequence database of PrefixSpan with pseudoprojection. However, according to Figure 11, Separator Database still benefits PrefixSpan with pseudoprojection.

PrefixSpan with pseudoprojection, either with Separator Database or without Separator Database, runs slower for larger datasets, as shown in Figure 12. The main reason is that larger datasets means more sequences to handle.

Nevertheless, Separator Database still consistently improves the runtime of PrefixSpan with pseudoprojection. Even in larger dataset size, PrefixSpan with pseudoprojection with Separator Database is more scalable than PrefixSpan with pseudoprojection without Separator Database.

Furthermore, based on Figure 13, PrefixSpan with pseudoprojection with Separator Database clearly wins against PrefixSpan with pseudoprojection without Separator Database when the customers buy more distinct items. It is plausible since in larger transaction length, checking on in-memory sequence database is much more frequent than checking on Separator Database. This surprising ratio of improvement shows that when the customer prefer to buy more distinct items in one basket, Separator Database enormously benefits the performance of PrefixSpan with pseudoprojection.

Based on Figure 14, although the improvement is only less than 20%, Separator Database saves about 100 MB of memory space. It shows that Separator Database affects not only the execution time but also the memory utilization.

Overall, the second series of experiments show that in terms of execution time and memory utilization, Separator Database improves sequential pattern mining using PrefixSpan with pseudoprojection, exponentially in some cases.

CHAPTER 5

CONCLUSIONS AND FUTURE WORKS

In this chapter, this thesis is summarized and future research on this area is presented.

5.1 Conclusions

The goal of this thesis is to improve PrefixSpan with pseudoprojection in terms of speed and memory utilization. This thesis proposes Separator Database and SPM-Tree Framework. The set of experiments is divided into two parts: (1) experiments to select suitable Collections for the proposed SPM-Tree Framework and (2) experiments to examine how the proposed Separator Database affects the performance of PrefixSpan with pseudoprojection.

The first set of experiments draws two major conclusions. First, using Java as a case study, ArrayList is the most suitable choice for storing Object, compared to Stack, Vector, and LinkedList. Second, ArrayList is the most suitable choice for storing integer data, compared to ArrayList, Stack, Vector, and LinkedList.

The second set of experiments shows the impact of Separator Database to the performance of PrefixSpan with pseudoprojection on four database characteristics. First, as the user-specified minimum support threshold plummets from 10% to 1%, Separator Database helps the performance of PrefixSpan with pseudoprojection to mine exponentially faster. Exponential growth of patterns found on smaller minimum support causes more candidate patterns need to be examined. Second, as the database gets denser, Separator

Database increases the performance of PrefixSpan with pseudoprojection to a constant ratio. Third, the growth of the number of customer elevates the performance of PrefixSpan with pseudoprojection. Fourth, as the number of distinct items bought by customer escalates, the performance of PrefixSpan with pseudoprojection also increases in exponential ratio. Separator Database affects not only the execution time but also the memory utilization. The ratio of improvement depends on the database characteristics. Based on the experiment from the previous chapter, although the improvement of Separator Database is only less than 20%, saving about 100 MB of memory presents a substantial improvement to PrefixSpan with pseudoprojection. Overall, the second series of experiments show that Separator Database improves sequential pattern mining using PrefixSpan with pseudoprojection, exponentially in some cases.

5.2 Future Works

There are many interesting issues that need to be studied further.

1. It is of interest to study the effect of Separator Database on mining generalized sequential patterns using the extension of PrefixSpan with pseudoprojection for mining constrained sequential patterns [15], [17], [18]. This study enables decision makers to put any sequential pattern mining constraints, such as time windows, minimum/maximum gaps, and taxonomy.
2. Often, large memory does not always exist in real life. Budget might be a hindrance for a decision maker to use decision support system such as sequential patterns. Thus, it is of interest to study mining sequential patterns using PrefixSpan with pseudoprojection with Separator Database on large database.

BIBLIOGRAPHY

-
- [1] D. Hand, H. Mannila, and P. Smith, "*Principles of Data Mining*", Cambridge: MIT Press, 2001.
 - [2] J. Han and M. Kamber, "*Data Mining: Concepts and Techniques*", San Fransisco: Elsevier Publishers, 2006.
 - [3] Q. Zhao and S. S. Bhowmick, "Sequential Pattern Mining: A Survey", CAIS, Nanyang Technological University, Singapore, Tech. Rep. 2003116, 2003.
 - [4] J. Han, J. Pei, and X. Yan, "Sequential Pattern Mining by Pattern-Growth: Principles and Extensions", in W. W. Chu and T. Y. Lin (eds.), *Recent Advances in Data Mining and Granular Computing (Mathematical Aspects of Knowledge Discovery)*, Springer, Verlag, 2004.
 - [5] R. Agrawal and R. Srikant, "Mining Sequential Patterns," in *Eleventh International Conference on Data Engineering (ICDE '95)*, 1995, pp. 3-14.
 - [6] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," in *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT '96)*, March 1996, pp. 3-17.
 - [7] F. Masegla, F. Cathala, and P. Poncelet, "The PSP Approach for Mining Sequential Patterns," *2nd European Symposium on Principles of Data Mining and Knowledge Discovery*, 1998, pp. 176-184.
 - [8] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning*, vol. 42, pp. 31-60, Jan/Feb 2001.
 - [9] J. Han and J. Pei, "Mining Frequent Patterns by Pattern-Growth: Methodology and Implications," *ACM SIGKDD Explorations Newsletter (special issue on scalable data mining algorithms)*, vol 2, pp.14-20, December 2000.
 - [10] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "FreeSpan: Frequent pattern-projected sequential pattern mining," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000, pp. 355-359.

- [11] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, M.-C., Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," in *Proceedings of the Seventeenth International Conference on Data Engineering (ICDE '01)*, 2001, pp. 215-224.
- [12] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, M.-C., Hsu, "Mining Sequential Patterns by Pattern Growth : The PrefixSpan Approach", *IEEE Transaction on Knowledge and Data Engineering*, vol. 16, pp.1041-4347, November 2004.
- [13] M.-Y. Lin and S.-Y. Lee, "Fast Discovery of Sequential Patterns through Memory Indexing and Database Partitioning," *Journal of Information Science and Engineering*, vol 21, pp. 109-128, January 2005.
- [14] M. N. Garofalakis, R. Rastogi, and K. Shim, "SPIRIT: Sequential Pattern Mining with Regular Expression Constraints," *The VLDB Journal*, pp. 223-234, September 1999.
- [15] J. Pei, J. Han, and W. Wang, "Mining Sequential Patterns with Constraints in Large Databases," In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, 2002, pp 18-25.
- [16] M.-Y. Lin, S.-Y. Lee, and S.-S. Wang "DELISP: Efficient Discovery of Generalized Sequential Patterns by Delimited Pattern-Growth Technology," in *Proceedings of the Sixth Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, London, 2002, pp 198-209.
- [17] J.-D. Ren, Y.-B. Cheng, and L.-L. Yang, "An Algorithm for Mining Generalized Sequential Patterns," in *Proceedings of the Third International Conference on Machine Learning and Cybernetics*, 2004, pp 1288-1292.
- [18] J. Pei, J. Han, and W. Wang, "Constraint-based sequential pattern mining: the pattern-growth methods", *Journal of Intelligent Information Systems*, vol. 28, pp. 133-160, April 2007.
- [19] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets," in *Proceedings of SIAM International Conference on Data Mining (SDM '03)*, 1999, pp. 166-177.

- [20] D. Yuan, K. Lee, H. Cheng, G. Khrisna, Z. Li, X. Ma, Y. Zhou, J. Han, "CISpan: Comprehensive Incremental Mining Algorithms of Closed Sequential Patterns for Multi-Versional Software Mining", presented at Proceedings 2008 SIAM International Conference on Data Mining (SDM'08), Atlanta, GA, April 2008.
- [21] J. Wang, J. Han, and C. Li, "Frequent Closed Sequence Mining without Candidate Maintenance", *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, pp. 1042-1056, August 2007.
- [22] C. Chen, X. Yan, F. Zhu, and J. Han, "gApprox: Mining Frequent Approximate Patterns from a Massive Network", in *Proceedings of International Conference on Data Mining (ICDM'07)*, 2007, pp. 445-450.
- [23] F. Zhu, X. Yan, J. Han, and P. S. Yu, "Efficient Discovery of Frequent Approximate Sequential Patterns", in *Proceedings of International Conference on Data Mining (ICDM'07)*, 2007, pp. 751-756.
- [24] H. Cheng, X. Yan, and J. Han, "IncSpan: Incremental Mining of Sequential Patterns in Large Database", in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*, 2004, pp. 527-532.
- [25] J. Han, J. Pei, and X. Yan, "From Sequential Pattern Mining to Structured Pattern Mining: A Pattern-Growth Approach," *Journal of Computer Science and Technology*, vol. 19, pp. 257-279, May 2004.
- [26] R. Kosala and H. Blockeel, "Web Mining Research: A Survey," *ACM SIGKDD Explorations Newsletter*, vol. 2, pp 1-15, June 2000.
- [27] P. H. Wu, W.C. Peng, and M.S. Chen, "Mining Sequential Alarm Patterns in a Telecommunication Database," in *Proceedings of the VLDB 2001 International Workshop on Databases in Telecommunications II*, 2001, pp. 37-51.
- [28] K. Wang, Y. Xu, and J. X. Yu, "Scalable sequential pattern mining for biological sequences," in *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, 2004, pp. 178-187.
- [29] Dejan Jelovic, "Why Java Will Always Be Slower Than C++, retrieved on March 10, 2008, http://www.jelovic.com/articles/why_java_is_slow.htm.

- [30] Sun Microsystems, Inc., "Sun Delivers Next Version of the Java Platform", December 8, 1998, <http://www.sun.com/smi/Press/sunflash/1998-12/sunflash.981208.9.xml>.
- [31] Sun Microsystems, Inc., "Sun Releases Fastest Client-Side Java Platform to Date", May 8, 2000, <http://www.sun.com/smi/Press/sunflash/2000-05/sunflash.20000508.3.xml>.
- [32] Dr. Dobb's Journal, "Microbenchmarking C++, C#, and Java", July 1, 2005, <http://www.ddj.com/java/184401976?pgno=1>.
- [33] M. F. Hornick, E. Marcadé, S. Venkayala, "Java Data Mining: Strategy, Standard, and Practice," San Fransisco: Elsevier Publishers, 2007.
- [34] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," in *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993, pp. 207-216.
- [35] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *Proceedings of the Twentieth International Conference on VLDB*, 1994, pp. 487-499.
- [36] S. Parthasarathy, M. J. Zaki, and W. Li, "Memory Placement Techniques for Parallel Association Mining," In *Proceedings of Fourth International Conference on Knowledge Discovery and Data Mining*, 1998, pp. 304-308.
- [37] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record*, vol. 29, pp. 1-12, May 2000.
- [38] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A Frequent-Pattern Tree Approach," in *Proceedings 2000 ACM-SIGMOD Int. Conf. Management of Data (SIG-MOD '00)*, pages 1-12, Dallas, TX, May 2004.
- [39] The Apache Software Foundation, "Apache Commons Primitives", March 5, 2008, <http://jakarta.apache.org/commons/primitives/>.

APPENDIX A: EXPERIMENT DATA

Table VII. Execution time of Appending Integers (in second)

Number of Data	LinkedList	Vector	Stack	ArrayList	ArrayIntList
500,000	499.8	406.2	356.2	300	93.8
1,000,000	887.4	778.2	690.4	506.2	143.6
1,500,000	1294	934.4	943.4	703	200
2,000,000	Out of memory	1206.2	1212.8	912.4	243.6
2,500,000	Out of memory	1359	1331	1228	262.4
3,000,000	Out of memory	Out of memory	Out of memory	1450	331.4
3,500,000	Out of memory	Out of memory	Out of memory	Out of memory	356.6
4,000,000	Out of memory	Out of memory	Out of memory	Out of memory	365.6

Table VIII. Execution time of Retrieving Integers (in second)

Number of Data	LinkedList	Vector	Stack	ArrayList	ArrayIntList
500,000	> 1 hour	43.8	44	3.2	3.2
1,000,000	> 1 hour	75.2	87.4	12.4	12.4
1,500,000	> 1 hour	116	115.6	12.6	24.8
2,000,000	> 1 hour	156.4	149.8	18.6	25
2,500,000	> 1 hour	190.6	190.6	28.4	28
3,000,000	> 1 hour	Out of memory	Out of memory	24.8	34.6

Table IX. Memory Usage of Appending Integers (in MB)

Number of Data	LinkedList	Vector	Stack	ArrayList	ArrayIntList
500,000	19.4156	10.6521	10.6916	12.2068	3.7501
1,000,000	38.5043	21.3935	21.2448	21.9843	8.0394
1,500,000	57.6195	38.205	38.1979	32.822	11.9096
2,000,000	Out of memory	41.1572	41.1503	45.2433	17.7136
2,500,000	Out of memory	48.5269	48.5476	60.1097	17.7146
3,000,000	Out of memory	Out of memory	Out of memory	59.7118	26.4072
3,500,000	Out of memory	Out of memory	Out of memory	Out of memory	26.4082
4,000,000	Out of memory	Out of memory	Out of memory	Out of memory	26.4096

Table X. Memory Usage of Appending and Retrieving Integers (in MB)

Number of Data	LinkedList	Vector	Stack	ArrayList	ArrayIntList
500,000	> 1 hour	10.7216	10.6288	12.1984	3.7461
1,000,000	> 1 hour	21.4088	21.2421	21.9821	8.0396
1,500,000	> 1 hour	38.2154	38.2208	32.8212	11.91
2,000,000	> 1 hour	41.1577	41.1574	45.2423	17.7123
2,500,000	> 1 hour	48.5323	48.5239	60.0928	17.7137
3,000,000	> 1 hour	Out of memory	Out of memory	59.3504	26.4106
3,500,000	> 1 hour	Out of memory	Out of memory	Out of memory	26.4079
4,000,000	> 1 hour	Out of memory	Out of memory	Out of memory	26.4056

Table XI. Execution Time of Appending Objects (in second)

Number of Data	LinkedList	Vector	Stack	ArrayList
500,000	340.6	128.2	134.2	115.8
1,000,000	643.6	237.6	241	156.2
1,500,000	965.6	365.4	365.6	215.6
2,000,000	1187.6	440.4	437.6	268.6
2,500,000	1677.8	509.4	515.8	371.8
3,000,000	Out of memory	687.8	687.6	381.4
3,500,000	Out of memory	759	750	509.2
4,000,000	Out of memory	834.4	834.4	522
4,500,000	Out of memory	925	825.6	549.8
5,000,000	Out of memory	1000	1000	565.4
5,500,000	Out of memory	Out of memory	Out of memory	647
6,000,000	Out of memory	Out of memory	Out of memory	656.2

Table XII. Execution Time of Retrieving Objects (in second)

Number of Data	LinkedList	Vector	Stack	ArrayList
500,000	>1 hour	49.8	44	6.2
1,000,000	>1 hour	97	75.2	9.2
1,500,000	>1 hour	124.8	116	16
2,000,000	>1 hour	162.4	149.8	25.2
2,500,000	>1 hour	218.4	190.6	31
3,000,000	>1 hour	231.2	Out of memory	28.2
3,500,000	>1 hour	318.6	Out of memory	34.4
4,000,000	>1 hour	317.8	Out of memory	47
4,500,000	>1 hour	421.8	Out of memory	43.8
5,000,000	>1 hour	468.8	Out of memory	56.2
5,500,000	>1 hour	Out of memory	Out of memory	52.8
6,000,000	Out of memory	Out of memory	Out of memory	65.8

Table XIII. Execution Time versus Database Density (in second)

Number of Distinct Items	PP w/ Separator DB	PP w/o Separator DB	Ratio	Difference
700	354.031	809.687	2.287051134	455.656
1400	328.5	752.687	2.291284627	424.187
1600	290.281	653.906	2.252665521	363.625
1800	212.594	501.531	2.359102327	288.937
2000	262.672	438.047	1.667657763	175.375
2100	3261.688	5725.344	1.755331595	2463.656
2200	2943.672	5392.891	1.8320285	2449.219

Table XIV. Execution Time versus Database Size (in second)

Number of Customers	PP w/ Separator DB	PP w/o Separator DB	Ratio	Difference
100	343.969	695.125	2.020894325	351.156
200	724.344	1443.485	1.992816949	719.141
400	1337.531	2721.375	2.03462574	1383.844
600	2058.922	4181.422	2.359102327	2122.5
800	2960.281	6054.828	2.045355829	3094.547
1000	5269.828	12503.97	2.37274727	7234.142

Table XV. Execution Time versus Minimum Support Threshold (in second)

Minimum Support	PP w/ Separator DB	PP w/o Separator DB	Ratio	Difference
10%	13.078	36.437	2.786129378	23.359
8%	20.438	56.375	2.758342304	35.937
6%	45.797	137.438	3.001026268	91.641
4%	128.968	425.063	3.295879598	296.095
2%	770.391	3026.234	3.928179327	2255.843
1%	4410.953	26276.11	5.957014278	21865.157
0.5%	34249.078	263475.031	7.692908726	229225.953

Table XVI. Execution Time versus Average Transaction Length (in second)

Transaction Length	PP w/ Separator DB	PP w/o Separator DB	Ratio	Difference
2	44.86	94.25	2.100980829	49.39
4	148.891	384.86	2.584843946	235.969
6	566.641	1655.671	2.921904698	1089.03
10	4410.953	26276.11	5.957014278	21865.157

Table XVII. Memory Usage versus Average Transaction Length (in MB)

Transaction Length	PP w/ Separator DB	PP w/o Separator DB	Ratio	Difference
50	637.185173	732.6030502	1.149749054	95.4178772
100	648.3291397	733.4485703	1.131290459	85.11943054
200	648.808876	760.4503708	1.172071466	111.6414948
300	647.5511551	759.7374573	1.173247011	112.1863022
400	647.3678436	760.5272369	1.174799219	113.1593933