Efficient Frequent Pattern Mining from Big Data and its Applications

by

Fan Jiang

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

January 2017

Thesis advisor

Dr. Carson K. Leung

Author

Fan Jiang

Efficient Frequent Pattern Mining from Big Data and its Applications

Abstract

Frequent pattern mining is an important research areas in data mining. Since its introduction, it has drawn attention of many researchers. Consequently, many algorithms have been proposed. Popular algorithms include level-wise Apriori based algorithms, tree based algorithms, and hyperlinked array structure based algorithms. While these algorithms are popular and beneficial due to some nice properties, they also suffer from some drawbacks such as multiple database scans, recursive tree constructions, or multiple hyperlink adjustments. In the current era of big data, high *volumes* of a wide *variety* of *valuable* data of different *veracities* can be easily collected or generated at high *velocity* in various real-life applications. Among these 5V's of big data, I focus on handling high *volumes* of big data in my Ph.D. thesis. Specifically, I design and implement a new efficient frequent pattern mining algorithmic technique called B-mine, which overcomes some of the aforementioned drawbacks and achieves better performance when compared with existing algorithms. I also extend my B-mine algorithm into a family of algorithms that can perform big data mining efficiently. Moreover, I design four different frameworks that apply this family of algorithms to the real-life application of social network mining. Evaluation results show the efficiency and practicality of all these algorithms.

Table of Contents

Abstract ii
Table of Contents v
List of Figures vi
List of Tables viii
Publications x
Acknowledgements xi
Dedication xiii

1 Introduction **1**
1.1 Thesis Statement 10
1.2 Problem Description and Thesis Contributions 11
1.3 Thesis Organization 13

2 Related Work **17**
2.1 Existing Frequent Pattern Mining Algorithms 18
2.1.1 Apriori Algorithm 18
2.1.2 FP-growth Algorithm 19
2.1.3 H-mine Algorithm 20
2.2 Big Data Mining 21
2.2.1 Big Data Mining with Parallel Computing and MapReduce . . 22
2.2.2 Big Data Mining with Data Compression 23
2.3 Social Network Mining 25
2.3.1 The Notion of "Following" Patterns 26
2.4 Summary 29

3 Frequent Pattern Mining with B-mine: The Fundamental Algorithm **31**
3.1 A New Data Structure 32
3.2 The B-mine Algorithm 33
3.3 Summary 42

4 Big Data Mining with B-mine: Parallel and MapReduce 43
4.1 PB-mine: The Parallel Version of B-mine 43
4.1.1 Data Structure Example of PB-mine Algorithm 46
4.1.2 Example of the Parallel Mining Algorithm PB-mine 47
4.2 MRB-mine: The MapReduce Version of B-mine 50
4.2.1 The First Set of Map-Reduce Functions in MRB-mine 52
4.2.2 The Second Set of Map-Reduce Functions in MRB-mine . . . 54
4.2.3 Subsequent Sets of Map-Reduce Functions in MRB-mine . . . 57
4.3 Summary 59

5 Big Data Mining with B-mine: Compression and Enhancement 61
5.1 The Compressed Bitmap Structure 61
5.1.1 Random Access to the Bit Vectors 62
5.1.2 Iterators 65
5.2 Big Data Mining with EB-mine 68
5.3 Summary 70

6 Real-life Application: Social Network Mining with B-mine 71
6.1 The Social Network Analysis Problem of Finding "Following" Patterns 71
6.1.1 The "Following" Relationships 72
6.1.2 Example #1 of "Following" Relationships 73
6.1.3 Example #2 of "Following" Relationships 74
6.1.4 Reduction to a Frequent Pattern Mining Problem 75
6.2 The FoP-miner Framework 76
6.2.1 The FoP-structure 76
6.2.2 Example of the FoP-structure 78
6.2.3 The Mining Process in FoP-miner 81
6.3 The ParFoP-miner Framework 89
6.3.1 Example of the FoP-structure for Parallel Social Network Mining 91
6.3.2 Example of the Parallel Mining Algorithm ParFoP-miner . . . 92
6.4 The BigFoP Framework 96
6.4.1 The First Set of Map-Reduce Functions in BigFoP 96
6.4.2 The Second Set of Map-Reduce Functions in BigFoP 99
6.4.3 Subsequent Sets of Map-Reduce Functions in BigFoP 102
6.5 The EFoP-miner Framework 105
6.6 Summary 110

7 Evaluation 111
7.1 B-mine and EB-mine Evaluation 111
7.1.1 Analytical Evaluation 111
7.1.2 Empirical Evaluation 113
7.2 PB-mine and MRB-mine Evaluation 115

7.2.1 PB-mine evaluation in the ParFoP-miner Framework 115
7.2.2 MRB-mine evaluation in the BigFoP Framework 116
7.2.3 EB-mine evaluation in the EFoP-miner Framework 120
7.3 Summary 122

8 Conclusions and Future Work **123**
8.1 Conclusions 123
8.2 Future Work 127
8.2.1 Uncertain Data Mining 128
8.2.2 Data Stream Mining 129

Bibliography **131**

List of Figures

1.1 An overview of this thesis work in terms of newly designed algorithms and frameworks. 14

2.1 A sample social network contains six users. Directed edges between users represent the "following" relationship between users. 28

3.1 The B-table created using example data shown in Table 3.1 34
3.2 The VI-List structure for Level 0 35
3.3 The HI-Counter structure for Level 0 35
3.4 The HICounterIncr() procedure for incrementing the counter. 36
3.5 The VIListComb() procedure for combining two VI-List structures. . 36
3.6 A level-1 B-table after cutting itemset {1} (i.e., a level-1 B-table formed for the 1-projected database) 37
3.7 The level-1 VI-List structure for itemset {1} 38
3.8 The level-1 HI-Counter structure for itemset {1} 38
3.9 A level-1+2 B-table after cutting itemset {1, 2} (i.e., a level-1+2 B-table formed for the {1, 2}-projected database) 39
3.10 The level-1+2 VI-List structure for itemset {1, 2} 39
3.11 The level-1+2 HI-Counter structure for itemset {1, 2} 39
3.12 The updated level-1 VI-List structure for itemset {1} (cf. the original level-1 VI-List structure in Figure 3.7) 40
3.13 A level-1+3 B-table after cutting itemset {1, 3} (i.e., a level-1+3 B-table formed for the {1, 3}-projected database) 41
3.14 The level-1+3 VI-List structure for itemset {1, 3} 41
3.15 The level-1+3 HI-Counter for itemset {1, 3} 41
3.16 The updated level-1 VI-List structure for itemset 1 (cf. the original level-1 VI-List structure in Figures 3.7 and 3.12) 42

6.1 A sample social network graph containing six users. Directed edges between users represent the "following" relationship between users. . 74
6.2 The Original SocialTable 106

6.3 The Compressed SocialTable 107
6.4 UserList for Level 0 108
6.5 A level-1 SocialTable after cutting followee 0 109
6.6 The level-1 VI-List structure for page 0 109

7.1 Experimental results on IBM synthetic datasets with 10K transactions. 115
7.2 Experimental results on IBM synthetic datasets with 100K transactions. 116
7.3 Experimental results on retail real-life datasets. 117
7.4 Experimental results of ParFoP-miner on social network datasets. . . 118
7.5 Experimental results of BigFoP on social network datasets. 119
7.6 Experimental results of EFoP-miner on social network dataset. 121

List of Tables

2.1 Example Database 18
2.2 User information extracted from Figure 2.1 28

3.1 B-mine Example Database 34

4.1 PB-mine Example Transaction Database 44
4.2 The B-table and its associated top-level HI-Counter & ParVI-List structures for PB-mine 47
4.3 B-tables and HI-Counter structures (in different processors) in PB-mine mining process domain items {A}, {B} and {C} 49
4.4 VI-List structures (in different processors) in PB-mine mining process domain item {A}, {B} and {C} 50
4.5 MRB-mine Example Transaction Database 51
4.6 The B-table Constructed from Table 4.5 51

5.1 A compressed bit vector of size 930. 63
5.2 A compressed bit vector of size 930. 65
5.3 The information for initialing iterator 66
5.4 The updated information for initialing iterator 67

6.1 The FoP-structure (i.e., SocialTable and its associated top-level HI-Counter & VI-List structures) 79
6.2 A FoP-structure for {social networking page A} 83
6.3 The FoP-structure for {social networking pages A, B} 84
6.4 An updated FoP-structure for {social networking page A} after mining {pages A, B} 85
6.5 The FoP-structure for {social networking pages A, C} 86
6.6 An updated FoP-structure for after mining the pages frequently followed together with {page A} 87
6.7 An updated FoP-structure for after mining the pages frequently followed together with {page B} 88

6.8 The FoP-structure (i.e., SocialTable and its associated top-level HI-Counter & ParVI-List structures) 92
6.9 SocialTables and HI-Counter structures for domain items {A}, {B} and {C} 94
6.10 VI-List structures for domain items {A}, {B} and {C} 95
6.11 The B-table Constructed from Figure 6.1 96

List of Publications

This is a list of publications that are closely related to this thesis, ordered by time.

- Mining Interesting "Following" Patterns from Social Networks. In *Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery* (*DaWaK 2014*), pages 308–319 [JL14b].

- A Business Intelligence Solution for Frequent Pattern Mining on Social Networks. In *Proceedings of the 2014 IEEE International Conference on Data Mining* (*ICDM 2014*) *Workshops*, pages 789–796 [JL14a].

- Big Data Analytics of Social Networks for the Discovery of "Following" Patterns. In *Proceedings of the 17th International Conference on Big Data Analytics and Knowledge Discovery* (*DaWaK 2015*), pages 123–135 [LJ15].

- Parallel social network mining for interesting 'following' patterns. *Concurrency and Computation: Practice and Experience*, 28(15), pages 3994–4012, 2016 [LJPP16].

- B-mine: Frequent Pattern Mining and Its Application to Knowledge Discovery from Social Networks. In *Proceedings of the 18th Asia-Pacific Web Conference* (*APWeb 2016*), pages 316–328 [JLZ16].

- Mining 'following' patterns from big sparse social networks. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (*ASONAM 2016*), pages 923–930 [DLJ16].

Acknowledgements

First, I would like to express my deepest gratitude to my Ph.D. research supervisor, Dr. Carson K. Leung. This thesis would not have been possible without his encouragement and academic support. Back when I was an undergraduate student, his enthusiasm and inspiration made me interest in the research area of data mining. After I finished my B.C.Sc. (Hons.) degree, Dr. Leung strongly encouraged me to stay in this research area, which gave me the opportunity to start my graduate study. Now, I look back, that was six years ago. During this six years, I finished my thesis based M.Sc. degree, taught some computer science courses in University of Manitoba as a sessional instructor, contributed more than 40 refereed research publications, and, most important, I learned so much about research from Dr. Leung. Therefore, without doubt, he is the first one on my list to appreciate.

Moreover, I would also like to thank my thesis examination committee members, Dr. Peter C.J. Graham, Dr. Xikui Wang and Dr. Osmar R. Zaïane, for their precious time to support my research, and also read my thesis. I would also like to thank Dr. Julien Arino to chair my Ph.D. oral defence on January 25, 2017.

I acknowledge the Database and Data Mining Laboratory, Department of Computer Science, and University of Manitoba for providing me a comfortable research environment, plenty of research equipment, and rapid technical support. I also want to thank my friends and lab members, Richard K. MacKinnon, Hao Zhang, Zhao Han, Peter Braun, Aaron M. Peddle, Edson M. Dela Cruz, as well as Oluwafemi A. Sarumi (a visiting graduate student from Nigeria), Mrigank Rochan (a graduate student from the Computer Vision Lab next door), and many others who helped me during my graduate studies at University of Manitoba. Particularly, I want to thank my fiancée,

Meiting Li, having her as companion in my life is the best thing that ever happened to me.

Last but not least, I thank my parents for their love, support, and understanding throughout my graduate study. It was them who made me realize my potential, and it was them who gave me the courage to chase my dream.

Fan Jiang

B.C.Sc. (Hons.), The University of Manitoba, Canada, 2010

M.Sc., The University of Manitoba, Canada, 2012

The University of Manitoba
January 2017

This thesis is dedicated to my parents and my fiancée.

Chapter 1

Introduction

Data mining aims to discover implicit, previously unknown, and potentially useful knowledge from data [AIS93]. One of the commonly used data mining tasks is *frequent pattern mining*, which finds frequently co-occurring items, events, or objects (e.g., frequently purchased merchandise items in shopper market baskets, frequently co-located conferences). Since the introduction of the research problem of frequent pattern mining, numerous frequent pattern mining algorithms [AS94, EZ06a, EZ06b, HPY00, PHL+01, SHS+00, Zak00, ZE05] and their extensions [CZLW15, FS16, MWW+15, NSK+16, WD15, XYX+15, ZCF+16] have been proposed.

One of the most famous frequent pattern mining algorithms is the Apriori algorithm [AS94] proposed by Aggarwal and Srikant. It applies a generate-and-test paradigm in mining frequent patterns in a level-wise, bottom-up fashion. In other words, the algorithm first generates candidate patterns of cardinality k (i.e., candidate k-itemsets) and tests if each of them is frequent (i.e., tests if its support or frequency meets or exceeds the user-specified minimum support threshold). Based on

these frequent patterns of cardinality k (i.e., frequent k-itemsets), the algorithm then generates candidate patterns of cardinality $k+1$ (i.e., candidate $(k+1)$-itemsets). This process is applied repeatedly to discover frequent patterns of all cardinalities. As one of the earliest designed frequent pattern mining algorithms, Apriori has some drawbacks. One of these drawbacks is that it requires k database scans to find a frequent pattern of cardinality k (i.e., a k-itemset), which leads to a large I/O cost.

The FP-growth algorithm [HPY00] is another famous frequent pattern mining algorithm. It uses an extended prefix-tree structure called a Frequent Pattern Tree (FP-tree) to capture the content of the transaction database. Unlike Apriori which scans the database K times (where K is the maximum cardinality of the discovered frequent patterns), FP-growth scans the database twice. The key idea of FP-growth is to recursively extract relevant paths from the FP-tree to form projected databases (i.e., collections of transactions containing some items), from which sub-trees (i.e., smaller FP-trees) capturing the content of relevant transactions are built. While FP-growth avoids the generate-and-test paradigm of Apriori (as FP-growth uses the divide-and-conquer paradigm), a drawback of FP-growth is that many FP-trees (e.g., for $\{a\}$-projected database, $\{a,b\}$-projected database, $\{a,b,c\}$-projected data-base, and so on) need to be built during the mining process. In other words, FP-growth requires lots of memory space.

To consider the mining process from a different angle, hyperlinked-array structure based mining algorithms (e.g., H-mine) were proposed [PHL$^+$01]. The H-mine algorithm also scans the database twice, but it captures the content of the transaction database in a hyperlinked-array structure called H-struct. During the mining process,

instead of recursively building sub-structures (e.g., smaller FP-trees as in FP-growth), H-mine reduces storage pressure by only recursively updating links in the existing H-struct to find frequent patterns. During the mining process, some array entries in the H-struct may contain K hyperlinks/pointers (where K is the maximum cardinality of the discovered frequent patterns), one hyperlink/pointer for each cardinality. A drawback of H-mine is that many of the hyperlinks/pointers need to be repeatedly updated during the mining process. This is a problem as these updates lengthen the runtime.

To recap, while the aforementioned frequent pattern mining algorithms are popular due to some advantages, they also suffer from a few drawbacks/disadvantages. For example, Apriori requires too many database scans and generates too many candidate patterns. As another example, FP-growth requires too much memory space to keep the global FP-tree (capturing the content of the original database) and all subsequent sub-trees (each capturing the contents of subsequent projected databases). As a third example, H-mine requires too many hyperlinks/pointers and too many update operations on these hyperlinks/pointers, and thus lengthen the runtime. Hence, some natural questions to ask are:

- Can I avoid the aforementioned drawbacks or disadvantages of these frequent itemset mining algorithms?
- Can I achieve better performance with a new algorithm?

Frequent Pattern Mining

As the first task and the first challenge of my Ph.D. thesis research, I successfully designed, implemented, and evaluated a new data structure and its corresponding

frequent pattern mining algorithm called B-mine, which gave a positive answer to the above questions. Based on the evaluation result, my new algorithm successfully overcame these aforementioned existing drawbacks and achieved better performance when compared with these existing algorithms.

The second task of my thesis was to apply my B-mine algorithm to big data mining. Big data [CSU13, Kej12, Mad12a] are everywhere nowadays. They are high-veracity, high-velocity, high-value, and/or high-variety data with volumes beyond the ability of commonly-used software to manage, query, and process within a tolerable elapsed time. These high volumes of valuable data can be easily collected or generated at a high velocity from different data sources (which may lead to different data formats) in various real-life applications such as bioinformatics, graph management, sensor and stream systems, smart worlds, social networks, as well as the Web [CBS13, JKL08, Mad12a, LJ15]. Characteristics of these big data can be described by the following "5 Vs":

1. Veracity, which focuses on the quality of data (e.g., uncertainty, messiness, or trustworthiness of data);
2. Velocity, which focuses on the speed at which data are collected or generated;
3. Value, which focuses on the usefulness of data;
4. Variety, which focuses on differences in types, contents or formats of data; and
5. Volume, which focuses on the quantity of data.

Embedded in the big data (e.g., biological data, medical images, streams of advertisements, surveillance videos, business transactions, financial charts, social media

data, web logs, texts and documents) are rich sets of useful information and knowledge. Due to the "5 Vs" characteristics of big data, new forms of algorithm are needed for managing, querying, and processing these big data so as to enable enhanced decision making, insight, process optimization, data mining and knowledge discovery. This drives and motivates research and practices in data science, which aim to develop systematic or quantitative data analytic algorithms to analyze (e.g., inspect, clean, transform, and model) and mine big data.

Big data analytics [KNR13, LH13, LJ14, LMJ14b, LML+16, SAM15, WKGL16] incorporates various techniques from a broad range of fields, which include cloud computing, data mining, machine learning, mathematics, and statistics. Various approaches ranging from mathematical models to approximation models, from resource-constrained paradigms to memory-bounded methods can be applied in cloud computing environments. Over the past few years, algorithms for handling big data according to a "systematic" view of the problem (e.g., MapReduce algorithms) have been gaining momentum.

Big Data Mining of Frequent Patterns

As the second part of my thesis, I applied my algorithm B-mine to big data mining focusing on analyzing big volumes of data (the 5th "V"). I further enhanced my B-mine algorithm in two research directions: one with parallel and MapReduce methods, another with a data compression method.

MapReduce [DG08] is a high-level programming model for handling high volumes of big data by using parallel and distributed computing [CLM14, LMJ14a, Zak99], sometimes on clouds [IJYZ12, IZ12, WWX15], which consist of a master node and

multiple worker nodes. As implied by its name, MapReduce involves two key functions: "map" and "reduce" functions commonly used in functional programming languages such as LISP for list processing:

1. The *mapper* applies a mapping function to each value in the list of values and returns the resulting list;

2. The *reducer* applies a reducing function to combine all the values in the list of values and returns the combined result.

An advantage of using the MapReduce model and an associated software framework is that users only need to focus on (and specify) these "map" and "reduce" functions without worrying about implementation details for the following:

- handling machine failures,
- managing inter-machine communication,
- partitioning the input data, or
- scheduling and executing the program across multiple machines.

On the other hand, as the second big data research direction, data compression can be applied to my algorithm. As "Volume" is one of the most important "V"s of big data, it is important to find a method that can compress the data as much possible so that an algorithm can process as much data as possible in a given period of time. In my thesis, other than implementing a parallel and MapReduce method for big data mining, I also implemented a data compression method for my algorithm.

Social Network Mining

The first two parts of my Ph.D. thesis were to (1) design an efficient frequent pattern mining algorithm, and (2) apply it to big data mining. Other than these two tasks, one other aspect of my research, that I believe is very important, is to see how it can be used to solve some real-life problems. Thus, as the third task of my Ph.D. thesis, I applied B-mine to social network data.

Social networks are generally made of social entities (e.g., individuals, corporations, collective social units, or organizations) that are linked by some specific types of interdependencies (e.g., kinship, friendship, common interest, beliefs, or financial exchange). A social entity is connected to another entity as his next-of-kin, friend, collaborator, co-author, classmate, co-worker, team member, and/or business partner, etc. Big data analytics of social networks computationally facilitates social studies and analysis of human-social dynamics in these big data networks, as well as designs and uses information and communication technologies for dealing with social context.

In the current era of big data, various social networking sites or services–such as Facebook, Google+, LinkedIn, Twitter, and Weibo [RG15, RZL15]–are commonly in use. As an example, Facebook users can create a personal profile, add other Facebook users as friends, exchange messages, and join common-interest user groups creating social relationships. The number of (mutual) friends may vary from one Facebook user to another. It is not uncommon for a user A to have hundreds or thousands of friends. Note that, although many of the Facebook users are linked to some other Facebook users via their mutual friendship (i.e., if a user A is a friend of another user B, then B is also a friend of A), there are situations in which such a relationship is

not mutual. To handle these situations, Facebook added the functionality of "follow", which allows a user to subscribe or follow public postings of some other Facebook users without the need of adding them as friends or being added as a friend. So, for any user C, if many of his friends followed some individual users or groups of users, then C might also be interested in following the same individual users or groups of users. Furthermore, the "like" button allows users to express their appreciation of content such as status updates, comments, photos, and advertisements. All of these actions result in social interaction data.

Similarly, Twitter users can read the tweets of other users by "following" them. Relationships between social entities are mostly defined by following (or subscribing) each other. Each user (social entity) can have multiple followers, and follows multiple users at the same time. The follow/subscribe relationship between follower and followee is not the same as the friendship relationship (in which each pair of users usually know each other before they setup the friendship relationship). In contrast, in the follow/subscribe relationship, a user D can follow another user E while E may not know D in person (e.g., we may follow the Royal family or our Prime Minister, but the Royal family members or the Prime Minister may not know us). This creates a relationship with direction in a social network. We use D $\rightarrow$ E to represent the follow/subscribe (i.e., "following") relationship that D is following E.

In recent years, the number of users in popular social networks has grown rapidly (as on June 2016, over 1.71 billion monthly active users for Facebook [fac], over 313 million monthly active users for Twitter [twi], and over 282 million active users for Weibo [wei]). This massive number of users creates an even more massive number

of "following" relationships. Hence, some natural questions to ask are:

1. For this huge amount of data containing user relationships and activities, can data mining techniques be applied? In response, over the past few years, several data mining algorithms and techniques [CLM14, JL13, LCJ13, LM14, ZZG+13] have been proposed. Many of them [KYWM15, LT12] are applicable to mine social networks (e.g. detection of communities [HSB+15, LXWC15, YDW+15], sub-graph mining [YYW+16], identification of decision makers [YCG16], as well as discovery of popular friends [JLLP14, LT12], influential friends [LTC14] and strong friends [TLC14]). For my Ph.D. research, I focus on the data mining task of finding frequent patterns [HPY00]-specifically, frequently following patterns.

2. Do these "following" relationships appear in certain ways/trends? For example, users who follow the twitter of SONY PlayStation are usually also following Microsoft XBOX at the same time. There could be other interesting hidden patterns in these huge amounts of following relationships. Discovering these patterns will greatly benefit both the social network users as well as service providers like Facebook, Twitter, or Weibo.

3. How can these interesting "following" patterns be discovered? How to discover these frequently followed individuals or groups? Manually going through the entire social networks is clearly impractical due to the huge amount of social data. A more systematic approach is needed. In response to the above questions, I applied my newly designed algorithm B-mine to efficiently discover frequently following patterns in social networks.

Overall, the **key challenges** of my thesis research include the following. First, I need to identify the drawbacks/disadvantages of existing frequent pattern mining algorithms and come up with space- and time-efficient algorithms that overcome these drawbacks/disadvantages while achieving better performance. Second, since a time-efficient algorithm usually requires more memory/disk space consumption while a space-efficient algorithm usually takes a long runtime, I need to keep a good balance/trade-off between the space- and time-efficiency requirements. To do so, a new data structure and its corresponding data mining algorithm are needed. Third, to illustrated usefulness, I need to apply this new algorithm to big data mining, which involving re-design the algorithm so that it can analyze big volumes of data. Fourth, applications of mining algorithms to real-life situations is also challenging due to the density and distribution of real-life data (e.g., finding frequent following patterns in social network dataset). My algorithm must have the ability to handle different types of data with different characteristics.

1.1 Thesis Statement

Motivated by the problems and solutions from the previous section, my thesis statement is to design and develop an algorithm for efficient frequent pattern mining, which includes the following:

- a new space- and time-efficient data structure for capturing data, as well as an algorithm that mines frequent patterns from such a new data structure, which both serve as solutions that overcome some drawbacks of existing algorithms and improve performance when compared with the existing algorithms;

- extensions to the above data structure and mining algorithm for big data mining which involve significant modifications (e.g., parallel/MapReduce and data compression) so that the new algorithms can process large volumes of data; and
- real-life applications of the developed data mining algorithms to practical problems (e.g., social network mining).

1.2 Problem Description and Thesis Contributions

Recall that, to mine frequent patterns, there are some existing methods (e.g., Apriori, FP-growth, H-mine, etc.). While they are popular and beneficial due to some good properties, they also suffer from some drawbacks. Based on this, some logical questions are:

1. Can we overcome these existing drawbacks and achieve better performance?
2. How to design a new data structure and a corresponding algorithm that will help us improve the performance?

Furthermore, since the big data mining is another challenge of my thesis, there are more questions that need to be answered:

3. Is the new algorithm applicable for big data mining?
4. What modifications to the new algorithm are needed for processing large volumes of data?
5. Can parallel/MapReduce be applied to the new algorithm?

6. Can data compression be applied to the new algorithm?

7. How well would the above two methods improve the performance?

8. Will the algorithm still be able to achieve better performance when compared with existing work for these extended research areas?

And finally, to work with social network data, three more questions need to be answered:

9. Is the new algorithm applicable to real-life problems (e.g., social network mining)?

10. What problem can it be applied to solve?

11. How effective can we solved this problem?

To response to the questions 1 and 2, I designed and implemented a new data structure, the B-table, and its corresponding mining algorithm B-mine. The evaluation results show that B-mine successfully overcame most of the aforementioned existing drawbacks and achieved better performance when compared with some existing algorithms (e.g., Apriori, FP-growth, and H-mine). For questions 3 to 8, I designed two improved versions of B-mine: (1) the parallel/MapReduce version of B-mine and (2) a compressed version of B-mine called EB-mine. The evaluation results showed that both these versions are able to handle significantly larger amount of data and even further improve the performance. Finally, for questions 9 to 11, B-mine was applied to social network mining. More specifically, B-mine was successfully applied to find frequent following patterns.

Hence, the key contributions of my Ph.D. thesis are:

1. The design and implementation of a new data structure and its corresponding frequent pattern mining algorithm to overcome many existing drawbacks or disadvantages from existing algorithms;

2. The analytical and experimental evaluation of the new algorithm;

3. The extension of my technique to big data mining using both parallel (or MapReduce) and data compression; and

4. Application of the newly designed algorithm to a real-life application (e.g., Social network mining).

Figure 1.1 shows a visualization of my thesis work in terms of newly designed algorithms and frameworks. The left circle represents algorithms for frequent pattern mining, they are: original B-mine (Chapter 3), PB-mine (Section 4.1), MRB-mine (Section 4.2). On the other hand, in the right circle, there are four frameworks that I designed for social network mining, they are: FoP-miner Framework(Section 6.2), ParFoP-miner Framework (Section 6.3), BigFoP Framework (Section 6.4), and EFoP-miner Framework (Section 6.5).

1.3 Thesis Organization

This thesis is organized as follows. The next chapter gives background information and related work, in which, I first describe some existing precise data mining algorithms. I will use a simple example to show the drawbacks/disadvantages of these

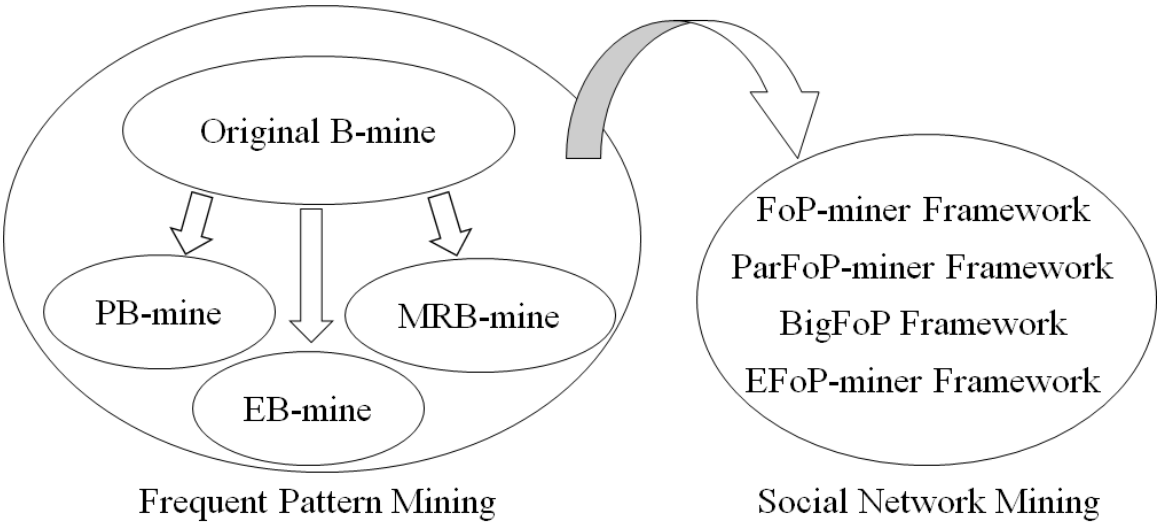

Figure 1.1: An overview of this thesis work in terms of newly designed algorithms and frameworks.

existing works. Then, I will introduce big data mining and background knowledge to my two solutions for handling big data. At the end of the next chapter, the idea of social network mining will be introduced together with the problem in social network mining that will be solved as part of my thesis work.

Afterwards, in Chapter 3, my fundamental proposed algorithm, B-mine, will be introduced in detail with examples. As a preview, B-mine uses a bitmap structure (called B-table) to store the transaction information, and two other data structures (HI-Counter and VI-List) to identify the target area of the bitmap structure during the mining process. This work has been published as a refereed full paper in the 18th Asia-Pacific Web Conference (APWeb 2016) [JLZ16].

In Chapter 4, I introduce two algorithms: PB-mine and MRB-mine. PB-mine enhances the original B-mine by applying parallel computing method. Because of the

characteristic of the bitmap structure (which I will discuss more in Chapter 4) that I designed in B-mine, data independency can be ensured. This gives me the power to enhance the original B-mine by parallelizing the mining process. This work has been published as one of the only four accepted articles among the top-7 DaWaK 2014 conference papers invited in a special issue of *Concurrency and Computation: Practice and Experience* [LJPP16]. On the other hand, MapReduce is another popular method for handling big data. In Section 4.2, I introduce MRB-mine, which is an algorithm enhanced from the original B-mine algorithm by applying MapReduce. This work has been published as a refereed paper in the 17th International Conference on Big Data Analytics and Knowledge Discovery (DaWaK 2015) [LJ15].

After that, in Chapter 5, EB-mine will be introduced. I am still working on handling big data, but from a different research direction–data compression. My original B-mine algorithm works fine with dense datasets, however, when working with sparse datasets, much storage space is "wasted" with "0"s, which leads to the idea of enhancement by data compression. In Chapter 5, I introduce EB-mine using an example dataset to show how the compression can be beneficial to the mining process, and how it can further improve the efficiency of the algorithm. This work has been published as a refereed paper in the International Symposium on Foundations and Applications of Big Data Analytics (ASONAM-FAB 2016) [DLJ16], and won the Best Paper award.

Chapter 6, introduces the real-life application of handling social network mining using my algorithm. There are four different frameworks that I designed to do social network mining: FoP-miner Framework [JL14b], ParFoP-miner Framework [LJPP16],

BigFoP Framework [LJ15], and EFoP-miner [DLJ16]. Each of these frameworks applies one of the aforementioned algorithms (B-mine and its extensions). They will all be explained with examples.

In Chapter 7, the evaluation results of all my work is presented and explained. Finally, I present my conclusions to my thesis work and discuss about future work in Chapter 8.

Chapter 2

Related Work

In this chapter, I provide background materials and related work that are relevant to my thesis. Recall from Chapter 1, my first task is to design a new algorithm, that can overcome several existing disadvantages of existing frequent pattern mining algorithms. In this chapter, I first introduce some of the existing algorithms and their corresponding drawbacks/disadvantages. After that, my next task is to extend my research to big data mining. Two enhancement directions will be applied to my algorithm, they are: (1) parallel method and MapReduce method, and (2) data compression method. In the second part of this chapter, I will introduce all relevant background knowledge for big data mining, including MapReduce and data compression. Furthermore, as the third task of my thesis is to apply my new algorithm to real-life application (e.g., social network mining), some necessary background will be introduced in the third part of this chapter.

2.1 Existing Frequent Pattern Mining Algorithms

To explain the mining process of Apriori [AS94], FP-growth [HPY00], and H-mine [PHL+01] in detail, let us consider an illustrative database with 6 transactions about 5 domain items (with item IDs 1, 2, 3, 4 and 5) as shown in Table 2.1. Let us set the user-specified minimum support (*minsup*) threshold to be 2, meaning that a pattern is frequent if it occurs at least twice in this transactional database.

Table 2.1: Example Database

Transaction ID	Contents (item IDs)
t_1	{2, 5}
t_2	{1, 3}
t_3	{1, 5}
t_4	{2, 3, 5}
t_5	{1, 2, 3}
t_6	{1, 2, 3, 5}

2.1.1 Apriori Algorithm

The Apriori algorithm first generates candidate 1-itemsets (potential frequent patterns) {1}, {2}, {3}, {4} and {5}. After scanning the database once, Apriori discovers frequent 1-itemsets {1}, {2}, {3} and {5}, each of which appears in four transactions. Based on these frequent 1-itemsets, Apriori generates candidate 2-itemsets {1,2}, {1,3}, {1,5}, {2,3}, {2,5} and {3,5}. Apriori then scans the database a second time to count the support (i.e., frequency) of these candidate 2-itemsets to find frequent 2-itemsets {1,2}, {1,3}, {1,5}, {2,3}, {2,5} and {3,5}, which appear twice, three times, twice, three times, three times and twice, respectively. Similarly, based on these frequent 2-itemsets, Apriori generates candidate 3-itemsets {1,2,3}, {1,2,5},

{1,3,5} and {2,3,5}. Apriori then scans the database a third time to count the support (i.e., frequency) of these candidate 3-itemsets to find frequent 3-itemsets {1,2,3} and {2,3,5}, which both appear twice. Based on these two frequent 3-itemsets, Apriori generates a candidate 4-itemset {1,2,3,5}, which is known to be infrequent because its subset {1,2,5} (or {1,3,5}) is not frequent. As observed from this example, the database is scanned multiple times during the entire mining process. Hence, Apriori may not be too time-efficient, especially when transactions in the database are long (which will lead to a high number of database scans when discovering frequent k-itemsets for high cardinality k).

2.1.2 FP-growth Algorithm

The FP-growth algorithm scans the database once to discover the frequent domain items 1, 2, 3 and 5. Then, FP-growth scans the database the second time to build an FP-tree to capture the contents of the above transactional database. From this global FP-tree (representing the entire data-base), FP-growth extracts appropriate tree paths to form an FP-tree for the {1}-projected database (i.e., transactions t_2={1,3}, t_3={1,5}, t_5={1,2,3} and t_6={1,2,3,5} containing item 1). From this FP-tree, FP-growth discovers frequent 2-itemsets {1,2}, {1,3} and {1,5}. FP-growth then builds an FP-tree for the {1,2}-projected database (i.e., transactions t_5={1,2,3} and t_6={1,2,3,5} containing both items 1 and 2), from which FP-growth discovers the frequent 3-itemset {1,2,3} and forms the {1,2,3}-projected database. Similarly, FP-growth builds an FP-tree for the {1,3}-projected database, also an FP-tree for the {1,5}-projected database (i.e., transaction t_3={1,5} containing both items 1 and

5 but not items 2 or 3). Note that, during the above mining process, FP-growth needs to keep multiple FP-trees (e.g., keeping the global FP-tree as well as the FP-trees for the {1,2}-projected database and the {1,2,3}-projected database, for a total of three FP-trees to be kept) in the memory at the same time. A similar mining process is then applied to the global FP-tree to form the {2}-projected database (i.e., transactions containing item 2 but not item 1), from which the {2,3}- and {2,3,5}-projected databases are formed. Again, these three FP-trees need to be kept in the memory. Note that, as the {2,5}-projected database is also formed from the {2}-projected database, a total of three more FP-trees need to be kept in the memory at the same time. Afterwards, a similar mining process is then applied to the global FP-tree to form the {3}-projected database (i.e., transactions containing item 3 but not items 1 or 2), from which the {3,5}-projected database is formed. As observed from this example, multiple FP-trees need to be kept in the memory during the mining process. Hence, FP-growth may not be too space-efficient, especially when transactions in the database are long (which may lead to a high number of coexisting FP-trees when discovering frequent k-itemsets for high cardinality k). For this high number of FP-trees, the runtime may also be high due to the traversal of a high number of FP-trees for forming a high number of projected databases. As such, FP-growth may also not be too time-efficient.

2.1.3 H-mine Algorithm

H-mine uses only one (global) H-struct to capture the contents of the original transactional database. Different items within each transaction are linked (e.g., items

1, 2, 3, and 5 are linked within transaction 6). During the mining process, the same k-itemsets in different transactions are also linked by hyperlinks/pointers. For example, H-mine first links item 1 in transactions 2, 3, 5 and 6 (in the process of discovering frequent 2-itemsets {1,2}, {1,3} and {1,5}), then links itemset {1,2} in transactions 5 and 6 (in the process of discovering frequent 3-itemsets {1,2,3} and {1,2,5}), and also links itemset {1,2,3} in transactions 5 and 6. In other words, multiple hyperlinks/pointers need to be updated during the mining process. Hence, H-mine may not be too time-efficient, especially when the database is large and when transactions in the database are long (which both may lead to a high number of hyperlinks/pointers) when discovering frequent k-itemsets for high cardinality k. For this high number of hyperlinks/pointers, it may become complicated to keep track of correct hyperlinks/pointers and to traverse the hyperlinks/pointers.

2.2 Big Data Mining

High volumes of valuable data (e.g., web logs, documents, business transactions, banking records, financial charts, medical images, surveillance videos, as well as streams of marketing, telecommunication, biological, life science, and social media data) can be easily collected or generated from different sources, in different formats, and at high velocity in many real-life applications in modern organizations and society. This leads us into the new era of *big data* [Mad12b], which refer to high-veracity, high-velocity, high-value, and/or high-variety data with volumes beyond the ability of commonly-used software to capture, manage, and process within a tolerable elapsed time. This drives and motivates research and practices in data science which aims

to develop systematic or quantitative processes to analyze and mine big data for continuous or iterative exploration, investigation, and understanding of past business performance so as to gain new insight and drive science or business planning. By applying big data analytics and mining (which incorporates various techniques from a broad range of fields such as cloud computing, data analytics, data mining, machine learning, mathematics, and statistics), data scientists can extract implicit, previously unknown, and potentially useful information from big data.

2.2.1 Big Data Mining with Parallel Computing and MapReduce

Over the past few years, researchers have used a high-level programming model-called *MapReduce* [DG08]–to process high volumes of big data by using parallel and distributed computing on large clusters or grids of nodes (i.e., commodity machines) or in clouds, which consist of a master node and multiple worker nodes. As implied by its name, MapReduce involves two key functions: (i) the Map function and (ii) the Reduce function. Specifically, the input data are read, divided into several partitions (sub-problems), and assigned to different processors. Each processor executes the map function on each partition (sub-problem). The map function takes a pair $\langle key_1, value_1 \rangle$ and returns a list of $\langle key_2, value_2 \rangle$ pairs as an intermediate result, where (i) key_1 and key_2 are keys in the same or different domains and (ii) $value_1$ and $value_2$ are the corresponding values in some domains. Afterwards, these pairs are shuffled and sorted. Each processor then executes the reduce function on (i) a single key key_2 from this intermediate result $\langle key_2$, list of $value_2 \rangle$ together with (ii) the list

of all values that appear with this key in the intermediate result. The reduce function "reduces"–by combining, aggregating, summarizing, filtering, or transforming–the list of values associated with a given key key_2 (for all k keys) and returns a single (aggregated or summarized) value $value_3$, where (i) key_2 is a key in some domain and (ii) $value_2$ and $value_3$ are the corresponding values in some domains. An advantage of using the MapReduce approach is that users only need to focus on (and specify) these "map" and "reduce" functions–without worrying about implementation details for (i) partitioning the input data, (ii) scheduling and executing the program across multiple machines, (iii) handling machine failures, or (iv) managing inter-machine communication. Examples of MapReduce applications include the construction of an inverted index as well as the word counting of a document for data processing [DG08].

2.2.2 Big Data Mining with Data Compression

One of the popular methods for handling big data is data compression. More specifically, for my thesis, data compression will be applied to bit-vectors/bitmaps with 0s and 1s (More detail for the data structure of my algorithm will be introduced in Chapter 3). A popular technique for reducing the memory footprint of bitmap indices is *bitmap compression* [SW09]. Bitmap compression shortens the length of bitmap indices by representing long runs of 1s or 0s in some other way such that it can easily tell how long the run is, but often less bits are used for storage. When long runs of 1s and 0s are common in the index, compression can have a major effect on the amount of storage space required. Hence, we can expect to see some tangible benefit by compressing the bit vectors for social networking information.

A popular bitmap compression scheme is the word-aligned hybrid compression scheme [SW09], which was primarily developed to be used as database index. Its key idea can be described as follows. Long bitmaps are divided into w = 32-bit words, which are then defined as either (1) fill words that are all 1s or all 0s, or (2) literal words that consist of mixture of 1s and 0s. In a 32-bit word, the first bit is to use for distinguishing fill words from tail/literal words. Specifically, if the first bit is 1, it is a fill word. Otherwise (i.e., the first bit is 0), it is a literal word. For the fill word, the second bit is used to indicate whether it is a word is filled with purely 1s or purely 0s. Specifically, a second bit of 0 indicates the word is a 0-fill word with sequences of 0s. A second bit of 1 indicates the word is a 1-fill word with sequences of 1s. The remaining 30 bits are the binary representation of the numbers of groups of 31 consecutive zeros (or ones) in the sequences. Hence, the number of 0s (or 1s) in a fill-word can be as low as one group, and can be as high as 230 groups of 31 zeros (or ones), i.e., $31 \times 2^30 \approx 33$ billion consecutive zeros (or ones).

Example 2.1 *Consider a bitmap vector with 31 bits 100 0000 1100 1111 0001 0000 0010 0011*$_{(2)}$*. As this vector consists of a mixture of 1s and 0s, we set the first bit to 0 to denote that it is a* literal word*, but we do not compress the remaining 31 bits. We just store them uncompressed. In other words, the vector can be stored as* ***0*** *100 0000 1100 1111 0001 0000 0010 0011*$_{(2)}$*. For readability, this vector can be shown in hexadecimal than binary representation: 40CF1023*$_{(16)}$*.*

Example 2.2 *Consider another bitmap vector with all 31 bits of consecutive 0s. As this vector consists of only 0s, we set the first bit to 1 to denote that it is a fill word and set the second bit to 0 to denote that it is a 0-fill word. As there is only 1 group of*

31 consecutive zeros, we represent this 1 group by using the remaining 30 bits: 00 0000 0000 0000 0000 0000 0000 0001$_{(2)}$. *Hence, this vector can be compressed to become* ***1 0*** *00 0000 0000 0000 0000 0000 0000 0001*$_{(2)}$. *Again, for readability, this compressed vector can be shown in hexadecimal than binary representation: 80000001*$_{(16)}$.

Example 2.3 *Consider the third bitmap vector with 186 bits, which can be represented in hexadecimal notation:*

40CF1023 ***00000000 00000000 00000000 00000000*** *0025101A*$_{(16)}$.

This vector consists of six words, in which the first and last ones are literal words, whereas the intermediate four are 0-fill words (i.e., $4_{(10)} = 100_{(2)} = 4_{(16)}$ *groups of 31 consecutive zeros). The vector can be compressed into the following:*

40CF1023 ***80000004*** *0025101A*$_{(16)}$.

Here, $80000004_{(16)}$ = ***1 0*** *00 0000 0000 0000 0000 0000 0000 0100*$_{(2)}$.

2.3 Social Network Mining

Recall from Chapter 1, social networks are generally made of social entities (e.g., individuals, corporations, collective social units, or organizations) that are linked by some specific types of interdependency (e.g., kinship, friendship, common interest, beliefs, or financial exchange). In recent years, the number of users in famous social networks grown rapidly. This massive number of users creates an even more massive number of "following" relationships. Our goal is to discover interesting/popular "following" patterns.

2.3.1 The Notion of "Following" Patterns

In social network sites like Twitter or Weibo, social entities (users) are linked by "following" relationships (e.g., a user p follows another user q). Compared with the friendship relationships in Facebook, this "following" relationship is different in the way that, the "following" relationships are directional. For instance, a user A may be following another user B while B is not following A. This property increases the complexity of the problem for two reasons. First, the group of users followed by A (e.g., A→B, C, ...) may not be same group of users as those who are following A (e.g., ..., X, Y, Z→A). Due to asymmetry in the "following" relationships (cf. the symmetric friendship relationships), we cannot use the usual backtracking methods to determine the reverse relationships (i.e. we cannot determine whether or not E→A if we only know A→E). In this case, we need to check both directions to get the relationship between pairs of users. Second, in terms of computation, due to asymmetry in the "following" relationships, the computation time and space is doubled. For each pair of users, A→B (A follows B) does not automatically imply B→A. It requires double the amount of memory space to store the relationship between two users (i.e., storing two directional "following" relationships A→B and B→A), whereas one only needs to store one unidirectional friendship relationship A – B. This also means that, if there is a change in the dataset (e.g., A unfollows B), then we cannot remove the linkage between A and B because B may still be following A (cf. when A "unfriends" B, the friendship linkage between A and B disappears). As the number of Twitter users is growing explosively nowadays, the number of relationships between Twitter users is also growing. One of the important research problems with regard to this massive

amount of data is to discover frequent "following" patterns.

A "following" pattern is a pattern representing the linkages when a significant number of users follow the same combination/group of users. For example, users who follow the twitter feed of NBA (The National Basketball Association in US) also follow the twitter feed of Adam Silver (current NBA commissioner). If there are large numbers of users who follow the twitter feeds of both NBA and Adam Silver together, we can define this combination (NBA and Adam Silver) of followees as a frequent "following" pattern (i.e., a frequently followed group). This pattern can be used for many purposes (e.g., Twitter feed recommendation, friend recommendation, etc.).

Example of "Following" Patterns

Let us consider an example on a social network that is represented as a graph shown in Figure 2.1, in which (i) each node represents a user (i.e., a social entity) in the social network and (ii) the follow/subscribe relationships between pairs of users are represented by directed edges between the corresponding nodes. The arrow on an edge represents the "following" direction (e.g., Bob→Carol represents that Bob follows Carol, Alice↔Bob represents that Alice and Bob are following each other). Then, we aim to find frequent "following" patterns.

From Figure 2.1, as this is a very small dataset, we can manually extract user information as shown in Table 2.2:

Note that Table 2.2 has two columns: 1) User name, and 2) Followee List (list of users that are followed by the current user). From Table 2.2, it is not very obvious but

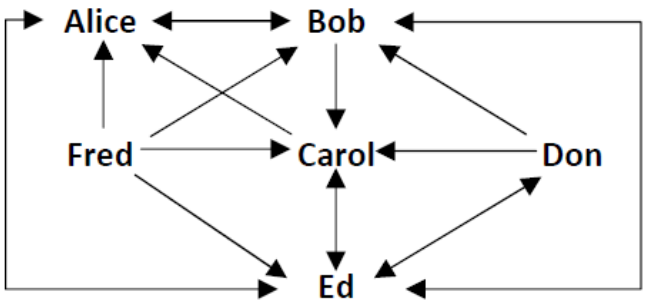

Figure 2.1: A sample social network contains six users. Directed edges between users represent the "following" relationship between users.

Table 2.2: User information extracted from Figure 2.1

User Name	**Followee List**
Alice	Bob, Ed
Bob	Alice, Carol, Ed
Carol	Alice, Ed
Don	Bob, Carol, Ed
Ed	Alice, Bob, Carol, Don
Fred	Alice, Bob, Carol, Ed

if we take a closer look, we can find some pairs of users that are usually followed by other users together. For example is that users Alice and Carol, are together followed by 3 users: Bob, Ed, and Fred. Another example, user Carol and Ed, are together followed by 3 users: Bob, Don, and Fred. This is a small database that only contains 6 users. If 3 of them (50%) are following a similar combination of other users, we can say this combination is a frequent "following" pattern. Furthermore, if there is a new user Gary added to this dataset, and the first followee that he chooses is Alice, according to our mining result, we can immediately recommend user Ed to Gary.

As we can see this is a small sample database with only 6 users, and the two examples that we see above are patterns with a pair of users (patterns with size 2).

Normally a social network will have millions of users. Mining frequent "following" patterns from millions of users can be really challenging.

2.4 Summary

In this chapter, I first introduced some classic frequent pattern mining algorithms for precise static datasets in Section 2.1. They are Apriori [AS94], FP-growth [HPY00], and H-mine [PHL+01]. The drawbacks of these existing algorithm were discussed: Apriori suffers from the large I/O cost caused by multiple database scans. FP-growth suffers from the construction and storage of multiple sub-structures. The performance of H-mine is largely rely on the complex hyperlinks update process. As a preview, I will propose a new algorithm that can overcome these drawbacks in the next chapter.

After that, I provided background information and reviewed literature on big data mining. I also introduced the most important challenge of big data mining. To extend my research to big data mining, two methods are needed: parallel and MapReduce (Section 2.2.1) and data compression (Section 2.2.2).

I also introduced some background knowledge of social network mining. Specifically, I introduced the notion of "following" patterns. For which, finding frequent "following" patterns is one of the real-life big data problem that I solved in my thesis research.

As a preview, I will propose my new algorithm for mining frequent patterns in the next chapter, which can overcome the aforementioned drawbacks of Apriori, FP-growth, and H-mine. Moreover, the extension to big data mining and the social

network mining will be introduced in Chapters 4 and 5, respectively.

Chapter 3

Frequent Pattern Mining with B-mine: The Fundamental Algorithm

Recall from Chapter 1, existing frequent pattern mining algorithms (e.g., Apriori [AS94], FP-growth [HPY00], H-mine [PHL+01], etc.) are beneficial due to some good properties, but they also suffer from some drawbacks such as multiple database scans, recursive tree constructions, or multiple hyperlink adjustments. In this chapter, I introduce my new algorithm, B-mine, which is designed to overcome these drawbacks from existing algorithms.

3.1 A New Data Structure

For discovering frequent patterns in transactional databases, in this section, I propose a new data structure to capture important contents in a transaction database. Here, I represent the contents of the transactions in a bit-wise tabular form. Such a table forms the basis of our data structure, which contains the following three key parts:

1. a **B-table** (Bitwise Table), which is the main bitmap based structure, in which each row contains information about one transaction in a database;
2. an **HI-Counter** (Horizontal Index Counter) vector of cardinality k (where k is the cardinality of the discovered patterns), which is an index counter list stored as a vector; and
3. a **VI-List** (Vertical Index List) structure of cardinality k, which is a two-dimensional data structure in which each row contains:
   (a) the prefix of the current row in VI-List structure,
   (b) a "column cutting index" which indicates the "cutting" points during the mining process, and
   (c) a list of "row cutting indices".

Given a transaction database (like what we will see in Table 3.1), to construct such a data structure, the following three steps are required:

1. For each row in the database, create a bitmap row with size M, where M is the number of domain items. Each column (represented by a bit) in the row

represents the items in each transaction. In other words, we put a "1" in the i^{th} bit (where $1 \leq i \leq M$) if the i^{th} item is present in the transaction; and put a "0" in the i^{th} bit (where $1 \leq i \leq M$) if the i^{th} item is absent from the transaction. When I repeat the aforementioned actions in this step, I get multiple bitmap rows to form a B-table.

2. Then, for each row r of the B-table, create a top-level VI-List structure. Specifically, I record the "row cutting index" (i.e., the column index for the first occurrence of a "1" bit in row r) in the VI-List structure.

3. Moreover, for each column c of the B-table, I also create a top-level HI-Counter structure. Specifically, I count the number of 1's in column c and put the count in the c^{th} position/entry of the HI-Counter structure.

In my new algorithm, to construct the B-table structure, the number of database scans can be decreased to only one, which is better than most existing frequent pattern mining methods. By using the bit-map data structure, I do not need to recursively construct sub-structures during the mining process. By using bit-operations in my algorithm, the mining process will be free from multiple and complex pointer operations.

3.2 The B-mine Algorithm

To explain the algorithm, let us consider the example in Table 3.1 (which reproduces the contents of Table 2.1). This example consists of 6 transactions and 5 domain items (with item IDs 1, 2, 3, 4 and 5).

Table 3.1: B-mine Example Database

Transaction ID	Contents (item IDs)
t_1	{2, 5}
t_2	{1, 3}
t_3	{1, 5}
t_4	{2, 3, 5}
t_5	{1, 2, 3}
t_6	{1, 2, 3, 5}

The B-table can be constructed by scanning the database only once to capture the contents of the transactional database. The resulting B-table is a bitwise data structure as shown in Figure 3.1. Note that, in Figure 3.1, each row represents a transaction. Though they are shown in the form of an array in Figure 3.1, in a real implementation, each 1 or 0 takes only one bit of memory.

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	0	0
3	1	0	0	0	1	0
4	0	1	1	0	1	0
5	1	1	1	0	0	0
6	1	1	1	0	1	0

Figure 3.1: The B-table created using example data shown in Table 3.1

Based on the B-table shown in Figure 3.1, we then construct both the VI-List and the HI-Counter structures of the highest level (i.e., top level, level 0) as shown in Figures 3.2 and 3.3, respectively.

Here, the HI-Counter structure is a vector that stores the count of each column of the B-table. For example, the count of "1" in column 2 (for item 1) of HI-Counter structure is 4, which means that item 1 appears in four transactions.

Row index	1-itemsets	Column Cutting Index	Row Cutting Indices
0	{1}	1	2, 3, 5, 6
1	{2}	2	1, 4

Figure 3.2: The VI-List structure for Level 0

	1	2	3	4	5	6
Level 0	4	4	4	0	4	0

Figure 3.3: The HI-Counter structure for Level 0

The VI-List structure, on the other hand, contains several rows corresponding to the first appearance of each item. For example, in Figure 3.2, the VI-List structure captures the information: (i) rows 2, 3, 5 and 6 (representing transactions 2, 3, 5 and 6) contain itemsets (i.e., four sets of items) that start with item 1; (ii) rows 1 and 4 (representing transactions 1 and 4) contain itemsets (i.e., two sets of items) that start with item 2 (but not item 1).

At this point, we can have the following observations:

Observation 3.1. The HI-Counter structure is an array of non-negative integers. The size of the HI-Counter structure is $(M - L)$, where M is the number of domain items and L is size of the current cardinality.

Observation 3.2. The size of each Row Cutting Indices in the VI-List structure is bounded above by N (i.e., at most N), where N is the total number of transactions.

Observation 3.3. The number of rows in the VI-List structure is bounded by the size of the current level of the HI-Counter structure (i.e., $M - L$).

Once the B-table, VI-List and HI-Counter structures are successfully constructed,

the mining process of B-mine can be performed. The process contains two major procedures, the HICounterIncr() and VIListComb(). See Figures 3.4 and 3.5.

```
HICounterIncr (b-table, HICounter[ ], VIList[ ])
{
  for each VIListRow in VIList[ ]
    rowIndice[ ] ← VIListRow.cuttingRowIndices()
    colIndex ← VIListRow.cuttingColIndices()
      for each rowIndex in rowIndice[ ]
        for each colIndex of b-table
          if the corresponding bit of b-table[rowIndex][colIndex] = 1
          then HICounter[colIndex]++
    Process final HICounter
    Create current level frequent pattern list
}
```

Figure 3.4: The HICounterIncr() procedure for incrementing the counter.

```
VIListComb (currLevelVIList, preLevelVIList[ ])
{
  currRow ← currLevelVIList.lastRow
    if (not all rows in preLevelVIList have been processed)
      for each row in prevLevelVIList
        if row.cuttingColIndex = currRow.cuttingColIndex
        then row.cuttingRowIndex += currRow.cuttingRowIndex
}
```

Figure 3.5: The VIListComb() procedure for combining two VI-List structures.

To explain the mining process, let us continue with our example from the point where the B-table was just constructed. Assume that the user specified minimum

support threshold (*minsup*) is 2, the mining process starts from the first row of the top-level VI-List structure as shown in Figure 3.2. In other words, let us start from the first row of our VI-List structure. Recall that each row of the VI-List structure consists of three parts:

1. The prefix of the current row in the VI-List structure (e.g., transaction 1);
2. a "column cutting index" (e.g., "Column 1"); and
3. an array of "row cutting indices" (e.g., "Rows 2, 3, 5 and 6").

By using the "column cutting index" and "row cutting indices", we can "cut" the B-table into some smaller pieces (as shown in Figure 3.6), to which the mining process for discovering frequent patterns with the item 1 or the 1-itemset {1} (as its prefix) can be applied. As shown in Figure 3.6, the original B-table is cut into the smaller section in the right hand side, namely, the level-1 B-table.

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	0	0
3	1	0	0	0	1	0
4	0	1	1	0	1	0
5	1	1	1	0	0	0
6	1	1	1	0	1	0

Figure 3.6: A level-1 B-table after cutting itemset {1} (i.e., a level-1 B-table formed for the 1-projected database)

It is important to note that the main B-Table does not change. The "cutting" process does not create new data structures. Instead, at this step, the B-mine algo-

rithm will focuses only on those interesting areas within the B-Table. For readability, uninteresting areas have been "faded out" Figure 3.6.

After this "cutting" process, the next step is to construct a level-1 HI-Counter and level-1 VI-List structures from the level-1 B-table. As shown in Figures 3.7 and 3.8, respectively, the newly constructed level-1 HI-Counter structure records all the column counts of the level-1 B-table. These counts are the frequency support values of all frequent patterns with prefix 1-itemset {1}. In this example, as we proceed to this step, we have frequency support values for frequent patterns: {1, 2}, {1, 3} and {1, 5}, which appear in 2, 3 and 2 transactions, respectively.

Row index	2-itemsets	Column Cutting Index	Row Cutting Indices
0	{1, 2}	2	5, 6
1	{1, 3}	3	2
2	{1, 5}	5	3

Figure 3.7: The level-1 VI-List structure for itemset {1}

	2	3	4	5	6
Level 1	2	3	0	2	0

Figure 3.8: The level-1 HI-Counter structure for itemset {1}

When we reach this step, we have the following two additional observations:

Observation 3.4. At this stage of the mining process, the current cardinality $k = 1$. Then, the size of the HI-Counter structure is $M - k = 6 - 1 = 5$.

Observation 3.5. The total number of Row Cutting Indices in the level-1 VI-List structure is the same as the number of Row Cutting Indices in the first row of level-0

VI-List structure (i.e. rows 2, 3, 5 and 6 are separated in to three subsets {5, 6}, {2} and {3}).

Based on this level-1 VI-List structure, we learn that we can construct a level-1+2 B-table (i.e. an updated B-table for itemset {1,2}) and its associated level-1+2 HI-Counter structure, and level 1+2-VI-List structure as shown in Figures 3.9, 3.10 and 3.11, respectively.

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	0	0
3	1	0	0	0	1	0
4	0	1	1	0	1	0
5	1	1	1	0	0	0
6	1	1	1	0	1	0

Figure 3.9: A level-1+2 B-table after cutting itemset {1, 2} (i.e., a level-1+2 B-table formed for the {1, 2}-projected database)

Row index	3-itemsets	Column Cutting Index	Row Cutting Indices
0	{1, 2, 3}	3	5, 6

Figure 3.10: The level-1+2 VI-List structure for itemset {1, 2}

	3	4	5	6
Level 1+2	2	0	1	0

Figure 3.11: The level-1+2 HI-Counter structure for itemset {1, 2}

Then, my algorithm returns the resulting 3-itemset {1,2,3} with frequency of 2. At this point, we observe that no more frequent results can be mined from the remainder

of the level-2 B-table for itemset {1,2}. Hence, the B-mine algorithm backtracks to the previous level. Afterwards, the algorithm checks if the previous VI-List structure has more rows. If no, return. If yes, the algorithm checks if all other rows in other previous VI-List structure have the same "cutting column index" as the current VI-List rows have. If yes, update the new row in the previous VI-List structure. Let us use the example to explain this process in details.

Continue with the mining process, after processing the first row of level-1 VI-List structure (as shown in Figure 3.7), there are two more rows remained in the level-1 VI-List structure. Notice that, as the column cutting index in level-1 VI-List[1] is the same as the level-1+2 VI-List[0], we need to update the level-1 VI-List[1] as shown in Figure 3.12.

Row index	2-itemsets	Column Cutting Index	Row Cutting Indices
~~**0**~~	~~**{1,2}**~~	~~**2**~~	~~**5, 6**~~
1	{1, 3}	3	2, **5, 6**
2	{1, 5}	5	3

Figure 3.12: The updated level-1 VI-List structure for itemset {1} (cf. the original level-1 VI-List structure in Figure 3.7)

We can observe from Figure 3.12 that, although the size of the "Row Cutting Indices" of the first level-1 VI-List structure is changed, the original property does not change, i.e., the total number of Row Cutting Indices in the level-1 VI-List structure is the same as the number of Row Cutting Indices in the first row of level-0 VI-List structure.)

To continue the mining process, we examine the {1,3}-projected database. Then,

the column "cutting" index is $3 + 1 = 4$, and rows 2, 5 and 6 are the "cutting" rows. Based on that, we can construct the corresponding level-1+3 B-table, HI-Counter and VI-List structures as shown in Figures 3.13, 3.14 and 3.15, respectively.

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	0	0
3	1	0	0	0	1	0
4	0	1	1	0	1	0
5	1	1	1	0	0	0
6	1	1	1	0	1	0

Figure 3.13: A level-1+3 B-table after cutting itemset {1, 3} (i.e., a level-1+3 B-table formed for the {1, 3}-projected database)

Row index	3-itemsets	Column Cutting Index	Row Cutting Indices
0	{1, 3, 5}	5	6

Figure 3.14: The level-1+3 VI-List structure for itemset {1, 3}

	4	5	6
Level 1+3	0	1	0

Figure 3.15: The level-1+3 HI-Counter for itemset {1, 3}

Afterwards, our B-mine algorithm backtracks to the level-1, and updates the level-1 VI-List structure as shown in Figure 3.16.

A similar step is applied to the {1,5}-projected database. Afterwards, our B-mine updates the level-0 VI-List structure to form the level-1 VI-List structure for itemset {2}. Consequently, we obtain our mining result, i.e., frequent patterns.

Row index	2-itemsets	Column Cutting Index	Row Cutting Indices
~~**0**~~	~~**{1, 2}**~~	~~**2**~~	~~**5, 6**~~
~~**1**~~	~~**{1, 3}**~~	~~**3**~~	~~**2, 5, 6**~~
2	{1, 5}	5	3, **6**

Figure 3.16: The updated level-1 VI-List structure for itemset 1 (cf. the original level-1 VI-List structure in Figures 3.7 and 3.12)

As I mentioned, B-mine can overcome several drawbacks from existing algorithms and achieve better performance. The evaluation results that support this statement will be shown in Chapter 7.

3.3 Summary

In this chapter, I proposed an efficient frequent pattern mining algorithm, B-mine, which includes a newly designed bitmap data structure and its corresponding mining algorithms. The bitmap data structure consists of three main parts: B-table, HI-Counter and VI-List structures. The mining algorithm utilizes the data structures to extract frequent patterns. An example of the B-mine mining process was also illustrated to show the detail of each steps. The evaluation of this algorithm will be presented in Chapter 7.

In the next two chapters (Chapters 4 and 5), I will introduce the big data mining component of this thesis work. Two different approaches are applied to my B-mine algorithm for big data mining, they are: a parallel approach and a compression approach. I will introduce both approaches in detail in the next two chapters.

Chapter 4

Big Data Mining with B-mine: Parallel and MapReduce

Recall from Chapter 1, one of the most important tasks in my thesis is to perform big data mining. In this chapter, I will introduce two algorithms, PB-mine and MRB-mine. Both of them are extended and enhanced from my original B-mine algorithm for big data mining. As a preview, PB-mine applies parallel computing (Section 4.1) while MRB-mine applies MapReduce (Section 4.2).

4.1 PB-mine: The Parallel Version of B-mine

The B-mine algorithm that I introduced in Chapter 3 is a serial algorithm. As we are living in the era of big data, high volumes of transaction data can be found everywhere. To speed up the mining process, I propose here a parallel algorithm called PB-mine for concurrent discovery of frequent patterns from a transaction database.

The key idea behind my parallel version of the B-mine algorithm can be described as follows. Based on the top-level B-table, the algorithm extracts from the transaction database the information relevant to each domain item and passes the information to available processors. Each processor then mines frequent patterns from this information in parallel. To make this happen, the algorithm extracts the information from the transaction database in such a way that the mining can be performed independently on each processor (i.e., one processor does not need to wait for results from another processor to start the mining). Specifically, based on the top-level B-table, each processor computes an $\{x\}$-projected B-table, which contains all transactions of domain item $\{x\}$, for every frequent/interesting domain item $\{x\}$. From the $\{x\}$-projected B-table, frequent patterns are then mined in a similar fashion as by my serial counterpart (i.e., B-mine algorithm). As a preview, my PB-mine algorithm extracts from the transaction database and passes (i) the information relevant to domain item {A} to Processor 1, (ii) the information relevant to domain item {B} to Processor 2, and (iii) the information relevant to domain item {C} to Processor 3.

Table 4.1: PB-mine Example Transaction Database

Transaction ID	Item List
t_1	{B, E}
t_2	{A, C, E}
t_3	{A, E}
t_4	{B, C, E}
t_5	{A, B, C, D}
t_6	{A, B, C, E}

A key question is: how to compute the $\{x\}$-projected B-table from a top-level B-table? Let's consider the transaction database in Table 4.1. If we apply the original

B-mine, it computes {A}-projected B-table based on the top-level VI-List structure containing information about domain item {A}, which appears in transactions 2, 3, 5, and 6. However, B-mine cannot compute {B}-projected B-table based on this top-level VI-List structure containing information about domain item {B} which appears in transaction 5 and 6 as it misses transactions 1 and 4, which are added to the list later after the mining process of domain item {A}. In other words, B-mine can only compute the {B}-projected B-table based on this *updated* top-level VI-List structure after the mining process of domain item {A}. However, this would mean that the processor for {B} could not start its mining until the completion of the mining task performed by the processor for {A}. Similar comments apply to the processor for {C}, which could not start its mining until the completion of the mining task performed by the processor for {B}.

To solve this problem, my PB-mine algorithm adapts the original VI-List structure to construct the following in addition to the usual (1) **B-table** and (2) **HI-Counter** structure:

3. a **Parallel-computation based Vertical Index List (ParVI-List)** structure,which is a two-dimensional data structure in which each row contains (a) a prefix of the current row in this ParVI-List structure indicating a group of domain items and (b) its corresponding list of transaction IDs based on the location of the relevant "1" bit (cf. first relevant "1" bit in the VI-List structure).

With the construction of (1) B-table, (2) HI-Counter and (3) ParVI-List structures, mining can be performed independently. This is because, each processor extracts relevant information from the top-level B-table to form an $\{x\}$-projected B-

table, which captures the transactions of frequent/interesting domain item $\{x\}$. The corresponding ParVI-List structure captures information about transactions for each of these domain items. As the mining on an $\{x\}$-projected B-table does not depend on the mining results on another $\{y\}$-projected B-table, the mining of frequent patterns can be performed independently among parallel processors.

Section 4.1.1 shows the data structure with this ParVI-List for the PB-mine, and Section 4.1.2 illustrates how our PB-mine algorithm discovers all frequent patterns from transaction database.

4.1.1 Data Structure Example of PB-mine Algorithm

Let us consider the same example transaction database shown in Table 4.1. This transaction database can be represented by the following structure shown in Table 4.2 for the process of PB-mine. While the B-table and HI-Counter structure are identical to what we had in B-mine, the ParVI-List structure here is different. Recall that the VI-List structure for B-mine was constructed based on the first occurrences of a "1" bit in each transaction. So, given that the first occurrences of a "1" bit in transaction 1 and 4 are in column B (domain item B), the VI-List structure contains information that the domain item B appears in transaction 1 and 4. Such a row in the VI-List structure will be updated later during the B-mine process. That is, after we process all frequent patterns with domain item A, the VI-List structure of domain item B will be updated to 1, 4, 5, 6 as these are those transactions that have B as either B as the first item, or B as the second item immediately after A (which, we already finished processing). On the other hand, in PB-mine, we create ParVI-List structure as shown

in Table 4.2. All this information is captured by ParVI-List structure without any need of updates. The reason is that the ParVI-List structure for parallel mining is constructed based on *all* occurrences (instead of just the *first* occurrence) of a "1" bit in each row.

Table 4.2: The B-table and its associated top-level HI-Counter & ParVI-List structures for PB-mine

B-table:	Items					
Transaction	A	B	C	D	E	F
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	1	0	0	0	1	0
4	0	1	1	0	1	0
5	1	1	1	1	0	0
6	1	1	1	0	1	0
HI-Counter:	4	4	4	1	5	0

ParVI-List:	
Items	List of transactions
{A}	2, 3, 5, 6
{B}	1, 4, 5, 6
{C}	2, 4, 5, 6
{E}	1, 2, 3, 4, 6

4.1.2 Example of the Parallel Mining Algorithm PB-mine

To illustrate my PB-mine process, let us continue with the above example. Let the user-specified frequency threshold (*minsup*) be 2, meaning that a pattern must appear in at least 2 transactions before we can call it *frequent* or *interesting*. With this setting, columns D and F can be ignored as their column sums in the top-level HI-Counter structure are below the user-specified frequency threshold. On the other hand, based on the column sums in this top-level HI-Counter structure, our mining algorithm found four interesting singleton patterns {A}, {B}, {C} and {E}.

Processors 1, 2 and 3 then use the top-level ParVI-List structure to extract all relevant rows from the B-table to compute the projected B-tables for domain items {A},

{B} and {C}, from which their corresponding HI-Counter and VI-List structures are constructed as shown in Table 4.3 and Table 4.4, respectively. Note that the ParVI-List structure is only constructed for the top-level. In all subsequent levels, the usual VI-List structures can be used unless projected B-table for such a level is further parallelized (say, for load balancing). Moreover, each processor only considers those relevant rows and columns, where relevant rows are indicated by the top-level ParVI-List structure. For instance, Processor 1, which computes the projected B-table for domain item {A}, considers only columns B, C & E (as D & F are infrequent/uninteresting) in transactions 2, 3, 5 and 6. Similarly, Processor 2, which computes the projected B-table for domain item {B}, considers only columns C & E in transactions 1, 4, 5 and 6. Processor 3, which computes the projected B-table for domain item {C}, considers only column E in transactions 2, 4, 5 and 6. Note that no processor is needed to compute the projected B-table for domain item {E} because the last domain item (i.e., {F}) is infrequent, thus no further frequent patterns can be generated after domain item {E}.

Once each processor has computed its B-table, the corresponding HI-Counter and VI-List structures can be easily constructed in the same way as in the original B-mine (Chapter 3). Based on these new HI-Counter structures, Processor 1 finds three frequent patterns namely, {A, B}, {A, C} and {A, E}). Processor 2 finds two frequent patterns (namely, {B, C} and {B, E}); Processor 3 finds one frequent pattern (namely, {C, E}). Based on the VI-List structures, Processor 1 computes the B-table for {A, B} to find a frequent pattern {A, B, C}. Afterwards, Processor 1 updates the VI-List structure for {A} by adding 5 & 6 as transactions to {A, C} and computes

Table 4.3: B-tables and HI-Counter structures (in different processors) in PB-mine mining process domain items {A}, {B} and {C}

B-table for {A}:	Items		
Transaction	B	C	E
2	0	1	1
3	0	0	1
5	1	1	0
6	1	1	1
HI-Counter for {A}:	2	3	3

B-table for {B}:	Items	
Transaction	C	E
1	0	1
4	1	1
5	1	0
6	1	1
HI-Counter for {B}:	3	3

B-table for {C}:	Items
Transaction	E
2	1
4	1
5	0
6	1
HI-Counter for {C}:	3

the B-table for {A, C} to find frequent pattern {A, C, E}. Similarly, Processor 2 computes the B-table for {B, C} to find {B, C, E} as the 13th frequent pattern.

Note that, in the above illustrative example, our PB-mine used 3 processors (for domain items {A}, {B} and {C}) because there are only 3+1 = 4 frequent/interesting domain items. With 4 frequent/interesting domain items ({A}, {B}, {C} and {E}), we do not need a processor for the last domain item (e.g., {E}) as no additional item

Table 4.4: VI-List structures (in different processors) in PB-mine mining process domain item {A}, {B} and {C}

VI-List for {A}:	
Items	Transactions
{A, B}	5, 6
{A, C}	2
{A, E}	3

VI-List for {B}:	
Items	Transactions
{B, C}	4, 5, 6
{B, E}	1

VI-List for {C}:	
Items	Transactions
{C, E}	2, 4, 5

would be frequent together with {E}. In general, for K frequent/interesting domain items, PB-mine requires only $K-1$ processors to achieve good efficiency. If there are surplus available processors, PB-mine can further parallelize some projected B-tables to take advantages of the extra processors. On the other hand, if there is a shortage of available processors, PB-mine can merge relevant transactions for multiple domain items.

4.2 MRB-mine: The MapReduce Version of B-mine

In this section, I will introduce a second big data mining algorithm, MRB-mine (MapReduce B-mine), which applies MapReduce to extract frequent patterns from

big data.

Recall from Section 2.2.1 that in the past few years MapReduce has become one of the most popular approaches for handling big data. In this section, I will introduce the detail steps of MRB-mine by illustrating the process using a simple transaction database.

Let us consider the following transaction database:

Table 4.5: MRB-mine Example Transaction Database

Transaction ID	Item List
t_1	{B, E}
t_2	{A, C, E}
t_3	{A, E}
t_4	{B, C, E}
t_5	{A, B, C, D}
t_6	{A, B, C, E}

The first step of MRB-mine is the same as the original B-mine: constructing a B-Table. Based on the transaction database in Table 4.5, we can build a B-Table as shown in Table 4.6:

Table 4.6: The B-table Constructed from Table 4.5

	A	B	C	D	E	F
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	1	0	0	0	1	0
4	0	1	1	0	1	0
5	1	1	1	1	0	0
6	1	1	1	0	1	0

4.2.1 The First Set of Map-Reduce Functions in MRB-mine

Abstractly, MRB-mine first applies a map function as follows:

$$\text{map}_1\text{: } \langle \text{The appearance of each ItemID in a transaction, current transaction ID} \rangle \mapsto \langle \text{Transaction ID, Item ID} \rangle, \tag{4.1}$$

in which the master node reads and divides the transaction database stored in the B-table into partitions. Specifically, the map_1 function can be specified as follows:

for each pair ⟨Transaction ID, Item ID⟩ in the transaction database **do**
  emit ⟨Transaction ID, Item ID, 1⟩.

This map function is applied to each pair of ⟨Transaction ID, Item ID⟩ in the transaction database, and results in a list of ⟨Transaction ID, Item ID, 1⟩ capturing all existing appearances of items in all transactions. See Example 4.1.

Example 4.1 *After applying the map_1 function to the B-table in Table 4.6, my MRB-mine algorithm returns a list containing ⟨1, B, 1⟩, ⟨1, E, 1⟩, ⟨2, A, 1⟩, ⟨2, C, 1⟩, ⟨2, E, 1⟩, ⟨3, A, 1⟩, ⟨3, E, 1⟩, ⟨4, B, 1⟩, ⟨4, C, 1⟩, ⟨4, E, 1⟩, ⟨5, A, 1⟩, ⟨5, B, 1⟩, ⟨5, C, 1⟩, ⟨5, D, 1⟩, ⟨6, A, 1⟩, ⟨6, B, 1⟩, ⟨6, C, 1⟩, and ⟨6, E, 1⟩.* ■

Afterwards, our big data analytics and mining solution MRB-mine applies a reduce function to group and count the number of transactions for each item, as well as to list these transactions for each item. More specifically, ⟨Transaction ID, Item ID, 1⟩ pairs from the map_1 function are shuffled and sorted. Each processor then executes the

reduce function on the shuffled and sorted pairs to count the number of transactions and list them for each item. To speed up this mining process, MRB-mine also allows users to specify a frequency threshold (*minsup*). By incorporating this user preference, MRB-mine returns (i) a list of transactions only for those frequent items (i.e., items that frequently appear in transactions) and (ii) the count for each item. In other words, MRB-mine applies the following reduce function:

$$\begin{aligned} \text{reduce}_1\text{:}\ & \langle \text{item, transaction list} \rangle \\ & \mapsto \text{list of } \langle \textit{frequent} \text{ item, transaction information} \rangle, \end{aligned} \tag{4.2}$$

with a detailed definition as follows:

for each item $\in \langle \_, \text{item}, \_ \rangle$ emitted by map_1 **do**
  set counter[item] = 0;
  set list[item] = {};
  for each transaction $\in \langle$transaction ID, item, 1$\rangle$ emitted by map_1 **do**
    counter[item] = counter[item] + 1;
    list[item] = list[item] $\cup$ {transaction ID};
  if counter[item] $\geq$ user-specified min frequency threshold
  then emit $\langle$item, counter[item], list[item]$\rangle$.

This results in (i) a list of transactions and (ii) its count for each *interesting/frequent* pattern. See Example 4.2.

Example 4.2 *Continue with Example 4.1. My MRB-mine applies the* reduce_1 *function with user-specified minsup of 2 and returns* $\langle A, 4, \{2, 3, 5, 6\}\rangle$, $\langle B, 4, \{1, 4, 5,$

6}⟩, ⟨C, 4, {2, 4, 5, 6}⟩, and ⟨E, 5, {1, 2, 3, 4, 6}⟩. Note that my MRB-mine does not return the lists for item D or F because their corresponding counters were lower than minsup.

To summarize, after applying the first set of map$_1$ *and reduce*$_1$ *functions, our MRB-mine has so far discovered four frequent patterns—in the form of* individual *frequent patterns—namely, {A}, {B}, {C} and {E}, who have supports 4, 4, 4 and 5 respectively. In other words, each of these four individual items appeared in at least 2 transactions.* ■

4.2.2 The Second Set of Map-Reduce Functions in MRB-mine

Thereafter, my MRB-mine applies a second set of map and reduce functions to mine frequent patterns in the form of *pairs* of items based on the results from the first set of map$_1$ and reduce$_1$ functions. For example, knowing that D and E are not frequent, it is guaranteed that any pairs of items (size-2 itemset) containing D or E is also not frequent. By making use of this knowledge, the search space for mining frequent patterns can then be pruned effectively. Specifically, the map$_2$ function, which returns ⟨transaction ID, $\{p\} \cup \{\text{item}\}$, 1⟩ for every transaction in the transaction list of each frequent pattern p, can be specified as follows:

$$\begin{aligned} \text{map}_2\text{:}\ & \langle \text{frequent pattern } p, \text{ its transaction information} \rangle \\ & \mapsto \langle \text{transaction, item pair} \rangle, \end{aligned} \tag{4.3}$$

with a detailed definition as follows:

for each $p \in \langle p, \_, \text{list}[p] \rangle$ emitted by reduce_1 **do**

  for each transaction $\in$ list$[p]$ **do**

    for each $\langle$transaction ID, item ID$\rangle$ in transaction database **do**

      if isRelevant(item, p)

      then emit $\langle$transaction ID, $\{p\} \cup \{\text{item}\}$, 1$\rangle$.

Here, isRelevant(item, p) is a Boolean function checking the relevance (e.g., consistence to the mining order) of item with respect to p. This results in lists of $\langle$transaction ID, item ID, 1$\rangle$, and a list for each frequent pattern p returned by the reduce_1 function. See Example 4.3.

Example 4.3 *Continue with Example 4.2. Recall that the first set of* map_1 *and* $reduce_1$ *functions returns four frequent patterns A, B, C and E. So, for frequent pattern A (appears in four transactions: 2, 3, 5 & 6), the* map_2 *function emits all* relevant *items of these four transactions:* $\langle 2, AC, 1\rangle$, $\langle 2, AE, 1\rangle$, $\langle 3, AE, 1\rangle$, $\langle 5, AB, 1\rangle$, $\langle 5, AC, 1\rangle$, $\langle 6, AB, 1\rangle$, $\langle 6, AC, 1\rangle$, *and* $\langle 6, AE, 1\rangle$. *Note that (i) items of transaction 4 are not emitted (because transaction 4 does not contain item A), and (ii)* $\langle 5, AD, 1\rangle$ *is irrelevant (because D is already known to be not frequent).*

Similarly, for frequent pattern B (appears in four transactions: 1, 4, 5 & 6), the map_2 *function emits all* relevant *items of these four transactions:* $\{\langle 1, BE, 1\rangle, \langle 4, BC, 1\rangle, \langle 4, BE, 1\rangle, \langle 5, BC, 1\rangle, \langle 6, BC, 1\rangle, \langle 6, BE, 1\rangle\}$. *Note that (i) items of transaction 3 are not emitted (because transaction 3 does not contain item B) and (ii)* $\langle 5, BD, 1\rangle$ *is irrelevant (because D is already known to be not frequent). More important to note is that (iii) patterns of the form* $\langle \_, AB, 1\rangle$ *(i.e.,* $\langle 5, AB, 1\rangle$, $\langle 6, AB,$

1⟩) are irrelevant with respect to p=B (because these patterns are already processed by the map₂ function).

Then, for frequent pattern C (appears in four transactions: 2, 4, 5 & 6), the map₂ function emits all relevant *items of these four transactions: {⟨2, CE, 1⟩, ⟨4, CE, 1⟩, ⟨6, CE, 1⟩}.*

Finally, for frequent pattern E (appears in five transactions: 1, 2, 3, 4 & 6), the map₂ function does not emit any items because there is no relevant *item in these five transactions.* ■

Similar to reduce$_1$, the reduce$_2$ function shuffles and sorts ⟨transaction, $\{p\} \cup$ {relevant item}, 1⟩ to find and count transactions for each item pair $P = (\{p\} \cup$ {relevant item}) as follows:

$$\begin{aligned} \text{reduce}_2\text{:}\ & \langle \text{item pair, list of common transactions} \rangle \\ & \mapsto \text{list of } \langle \textit{frequent} \text{ item pair, transaction information} \rangle, \end{aligned} \tag{4.4}$$

with a detailed definition as follows:

for each $P \in \langle \_, \text{item group } P, \_ \rangle$ emitted by map$_2$ **do**

set counter$[P] = 0$;

set list$[P] = \{\}$;

for each transaction $\in \langle$transaction, P, 1$\rangle$ emitted by map$_2$ **do**

counter$[P]$ = counter$[P]$ + 1;

list$[P]$ = list$[P] \cup$ {transaction};

if counter$[P] \geq$ user-specified min frequency threshold

then emit $\langle P$, counter$[P]$, list$[P]\rangle$.

This results in (i) a list of transactions and (ii) its count for each *interesting/frequent* item pair P.

Example 4.4 *Continue with Example 4.3. My MRB-mine algorithm applies the reduce$_2$ function with user-specified minimum frequency threshold = 2 transactions and returns* $\langle AB,\ 2,\ \{5,\ 6\}\rangle$, $\langle AC,\ 3,\ \{2,\ 5,\ 6\}\rangle$, $\langle AE,\ 3,\ \{2,\ 3,\ 6\}\rangle$, $\langle BC,\ 3,\ \{4,\ 5,\ 6\}\rangle$, $\langle BE,\ 3,\ \{1,\ 4,\ 6\}\rangle$, *and* $\langle CE,\ 3,\ \{2,\ 4,\ 6\}\rangle$. *In other words, after applying this second set of map$_2$ and reduce$_2$ functions, my MRB-mine algorithm discovered six frequent patterns—in the form of pairs of items—namely,* $\{A,B\}$, $\{A,C\}$, $\{A,E\}$, $\{B,C\}$, $\{B,E\}$ *and* $\{C,\ E\}$, *that have support values 2, 3, 3, 3, 3 and 3 respectively. In other words, each of these six item pairs appears in at least 2 transactions.* ■

4.2.3 Subsequent Sets of Map-Reduce Functions in MRB-mine

So far, my MRB-mine algorithm has found frequent patterns in the form of (i) *individual* frequent items as well as (ii) *pairs* of frequent items. MRB-mine then applies *similar* sets of map and reduce functions to find triplets, quadruplets, quintuplets and higher cardinality groups (i.e., k-tuplets for $k \geq 3$) of frequently patterns:

$$\begin{aligned} \text{map}_{k\geq 3}\text{:}\ \ &\langle\text{frequent item } (k-1)\text{-tuplet } P\text{, transaction information}\rangle \\ &\mapsto \langle\text{transaction, item } k\text{-tuplet}\rangle, \end{aligned} \tag{4.5}$$

with a detailed definition as follows:

for each $P \in \langle P, \_, \text{list}[P] \rangle$ emitted by reduce_{k-1} **do**

for each transaction $\in \text{list}[P]$ **do**

for each $\langle$transaction, item$\rangle$ in transaction database **do**

if isRelevant(item, P)

then emit $\langle \text{transaction}, P \cup \{\text{item}\}, 1 \rangle$.

Again, isRelevant(item, P) is a Boolean function checking the relevance (i.e., consistence to the mining order) of item with respect to P. Since reduce_2 can be considered as an instance of the $\text{reduce}_{k\geq 2}$ function, the latter can be defined in a way very similar to that for reduce_2 as shown below:

$$\begin{aligned} \text{reduce}_{k\geq 2}\text{:} \quad & \langle \text{item group, list of common transactions} \rangle \\ & \mapsto \text{list of } \langle \textit{frequent} \text{ item group, transaction information} \rangle, \end{aligned} \tag{4.6}$$

with a detailed definition as follows:

for each $P \in \langle \_, \text{item group } P, \_ \rangle$ emitted by map_{k-1} **do**

set $\text{counter}[P] = 0$;

set $\text{list}[P] = \{\}$;

for each transaction $\in \langle \text{transaction}, P, 1 \rangle$ emitted by map_{k-1} **do**

$\text{counter}[P] = \text{counter}[P] + 1$;

$\text{list}[P] = \text{list}[P] \cup \{\text{transaction}\}$;

if $\text{counter}[P] \geq$ user-specified min frequency threshold

then emit $\langle P, \text{counter}[P], \text{list}[P] \rangle$.

This results in (i) a list of transactions and (ii) its count for each *interesting/frequent*

item group P. See Example 4.5.

Example 4.5 *Continue with Example 4.4. For frequent item group AB (appears in two transactions: 5 & 6), the map_3 function emits three* relevant *items: $\{\langle 5, ABC, 1\rangle, \langle 6, ABC, 1\rangle, \langle 6, ABE, 1\rangle\}$. Then, for frequent item group AC (appears in three transactions: 2, 5 & 6), the map_3 function emits two* relevant *items: $\{\langle 2, ACE, 1\rangle, \langle 6, ACE, 1\rangle\}$. Similarly, for frequent item group BC (appears in three transactions: 4, 5 & 6), the map_3 function emits two* relevant *items: $\{\langle 4, BCE, 1\rangle, \langle 6, BCE, 1\rangle\}$.*

Afterwards, by applying the $reduce_3$ function, my MRB-mine algorithm discovers the following three frequent patterns $\{A, B, C\}$, $\{A, C, E\}$ and $\{B, C, E\}$: $\langle ABC, 2, \{5, 6\}\rangle$, $\langle ACE, 2, \{2, 6\}\rangle$, and $\langle BCE, 2, \{4, 6\}\rangle$.

Based on the results returned by the $reduce_3$ function, MRB-mine applies map_4 but returns nothing because there is no relevant quadruplet of frequent patterns. This completes the mining process for frequent patterns from our illustrative example transaction database. Note that key concepts and steps illustrated in this example are applicable to any big transaction database. ■

4.3 Summary

In this chapter, I proposed two algorithms, PB-mine and MRB-mine for big data mining of frequent patterns. Both algorithms were designed based on the original B-mine algorithm (introduced in Chapter 3). PB-mine applies parallel computing based on the unique characteristic (data independency can be easily found) that the data structure of B-mine has. It distributes the data into several computing processors without violating the data independency, and from there, all processors

can, together, perform in parallel to find frequent patterns from big data. On the other hand, as MapReduce is a well-known approach for big data mining, MRB-mine applies MapReduce to extract frequent patterns from big data. The evaluation of both PB-mine and MRB-mine will be presented in Chapter 7.

Chapter 5

Big Data Mining with B-mine: Compression and Enhancement

Recall from Chapter 1, to handle big data, I have two research directions. The first one, parallel computing and MapReduce were introduced in Chapter 4. In this chapter, I will introduce the second one, compression and enhancement.

5.1 The Compressed Bitmap Structure

While representing a transaction database with a bitwise representation seems logical, each list of domain items for a transaction can be long but sparse. For instance, one of the IBM Synthetic datasets that I used for evaluation purpose (more detail about this dataset and corresponding evaluations will be discussed in Chapter 7) contains 1000 domain items, while the average length of all transactions is less than 20 (Each row of the bitmap will therefore be 1000 bits in length, while only around

20 of those bits will be marked as 1).

To avoid storing a lot of 0s, compression schemes are needed. When using some compression techniques (e.g., run-length encoding (RLE)), the "compressed" bitmap for a dense portion of bitmap can be longer than the original uncompressed bitmap. Instead, I adopt the word-aligned hybrid (WAH) compression scheme [SW09], which was primarily developed to be used as database index. An advantage of using this hybrid compression scheme is that I do not need to compress everything. On the one hand, for the sparse portion of the bitmap, I can compress the portion to save space. On the other hand, for the dense portion of the bitmap, I can leave the portion untouched.

5.1.1 Random Access to the Bit Vectors

I propose the following compressed bitmap structure that helps facilitate frequent pattern mining from big data. This structure is composed of the following:

- An internal array containing n words, each of size w (e.g., $w = 32$ in the previous description);
- A small bit vector containing n bits, which I call a Fill Map. For each word at index i in the internal array, we set the fill map bit i to (a) 0 if is a literal word and (b) 1 if it is a fill word. The fill map indexing structure will be used to provide fast random access performance across the bit vector.

Consider the following illustrative example with a compressed bit vector of size 930, with "1" bits in positions 0, 15, 40, 50, 92, and 646. Our structure is composed of (a) index, (b) fill map, and (c) vector (as shown in Table 5.1).

Table 5.1: A compressed bit vector of size 930.

Index	0	1	2	3	4	5
FillMap	0	0	0	1	0	1
Array	00008001	00080200	40000000	80000011	04000000	80000009

One of the challenges when dealing with compressed bit vectors is that it is impossible to tell which bit positions are in a particular word without doing a traversal from either end and keeping track of where we are in terms of bit index.

When dealing with uncompressed bit vectors, one has the luxury of looking at a particular index, say 5, and knowing it contains [5w, 6w) bits where w is the size of (i.e., the number of bits in) a word. However, when dealing with the adapted WAH-compressed bit vector, it is difficult to look at a particular index because an adapted WAH-compressed bitmap vector may be a mixture of literal and fill words of varying sizes.

To alleviate this problem, I created the FillMap indexing structure. For efficiency, my WAH-compressed bit vector makes extensive use of CPU instructions "Find First Set", which includes the following:

- *Count Trailing Zeros (CTZ)* returns the number of zeroes between index 0 and the next bit equal to 1. We can observe that the number of zeroes and the index of the 1 bit is the same.

- *Count Leading Zeros (CLZ)* returns the number of zeroes between index $w1$ (where w is the word size) and the next bit equal to 1. It works exactly as CTZ except it works from the other end. This machine instruction was used in our WAH-compressed data structure for reverse iteration), which has wide

hardware support across different architectures.

These instructions, combined with simple masking of bits, make it possible to quickly find the index of fill words in the internal array. This enables the code to efficiently keep track of where we are in the bit vector.

To illustrate the usage of the fill map, let us continue with our example and try to find the value of bit at index 646 from our adapted WAH-compressed bitmap structure.

First, we call the CLZ function on the WAH-compressed structure. The CLZ function returns 3 because there are 3 zeros between index 0 and the next "1" bit in the FillMap. So, we can safely skip 3 literal words, which is equivalent to $3 \times (w-1)$ bits $= 3 \times (321) = 93$ bits. Then, there is a 0-fill word (as indicated by its first bit "1" in 1000 0000 ... 0000 0001 $0001_{(2)} = 80000011_{(16)}$) located at index 3 of the internal array. Extracting its fill count would give us $17_{(10)} = 11_{(16)} = 000\ 0000\ ...\ 0000\ 0001\ 0001_{(2)}$, which means up to and including index 3, we have visited bits 0 to $93 + 17w = 93 + 17 \times 31$, i.e., a total of 620 bits.

As $620 < 646$, we then move on to the next word at index 4. We call the CLZ function once again. We mask the fill map up to the position we have read so far. In this case, we need to AND the bits 0-3 with 0, before calling the CLZ function once again.

Next, a call to the CTZ function returns 5. Subtracting our current FillMap bit position 4 from this give us $54 = 1$, which tells us to skip 1 literal word (i.e., 31 bits). By doing so, we reach position $620 + 31 = 651 > 646$. Hence, we can now safely extract bit 646 from the literal word at index 4.

5.1.2 Iterators

One of the strengths of compressed bit vectors is that it provides very fast and efficient traversal of relevant bits; i.e., those that are equal to 1. Compression removes large gaps of consecutive zeroes (and ones) and brings the relevant bits closer together in a more compact and efficient form. To perform a traversal, we need to create an iterator that holds the following information:

- *Bit Index*, which keeps track of the current bit position in the traversal.
- *Array Index*, which keeps track of the current index in the internal array.
- *Lower Bound*, which keeps track of the lower bound bit index represented by the current word pointed at by the Array Index (e.g., if the current word contains bits 31-62, then the lower bound is equal to 31).
- *Upper Bound*, which keeps track of the upper bound bit index represented by the current word pointed at by the Array Index (e.g., if the current word contains bits 31-93 (i.e., its a fill word with fill count = 2), then the upper bound is equal to 93).

Let us revisit the aforementioned adapted WAH-compressed bit vector in Table 5.2.

Table 5.2: A compressed bit vector of size 930.

Index	0	1	2	3	4	5
FillMap	0	0	0	1	0	1
Array	00008001	00080200	40000000	80000011	04000000	80000009

To begin a forward traversal on the adapted WAH-compressed bit vector in Table 5.2, we initialize the iterator with the information shown in Table 5.3:

Table 5.3: The information for initialing iterator

Bit Index	0
Array Index	0
Lower Bound	0
Upper Bound	31

Here, (a) bit index is set to 0 because we are starting from the beginning, (b) array index is set to 0 because we are looking at the first word in the internal array, (c) lower bound is set to 0 because we are starting from the beginning, and (d) upper bound is set to 31 because the first word is a literal and thus contains bits 0-31. Note that, if the first word happens to be a fill word, then its value would have been lower bound + (*Fill Count* $\times$31).

There are three types of words that can be encountered when traversing an adapted WAH-compressed bit vector: (a) a literal word, (b) a 0-fill word, and (c) a 1-fill word:

- If a *literal word* is encountered, we can call the CTZ function (or the CLZ function in the reverse iteration) to extract the bit position of the next bit that is equal to 1.

- If a *0-fill word* is encountered, we simply move to the next word (because there are no bits equal to 1 in a 0-fill word by definition).

- If a *1-fill word* is encountered, we return the current Bit Index, and increment it by one until we reach the Upper Bound of the current word, then we move

to the next word.

When moving to the next word, we need to update the iterator as follows:

- We increment the internal array index by 1.
- We set the bit index and lower bound to the current upper bound.
- If the next word is a literal word, then we add 31 to the current upper bound.
- If the next word is a fill word, then we add Fill Count $\times 31$ to the current upper bound.

Let us continue with our example with Table 5.2 and Table 5.3.

After moving the iterator to the next word in the internal array, Table 5.3 is updated to Table 5.4.

Table 5.4: The updated information for initialing iterator

Bit Index	31
Array Index	1
Lower Bound	31
Upper Bound	63

Traversing the adapted WAH-compressed bit vector in this manner is very efficient, and only takes an amount of time proportional to the number of bits set to 1. This leads to several important implications:

- We can tell outright if an adapted WAH-compressed bit vector is empty.
- We can tell if there are no more elements left without checking all bit positions.
- It can be used with list-like and stack-like interface.

5.2 Big Data Mining with EB-mine

With our compressed bitmap structure described in the previous section, we can discover frequent patterns from big data.

Recall from Chapter 3 the data structure of B-mine has the following three key parts:

1. a B-table,
2. an HI-Counter structure, and
3. a VI-List structure.

In EB-mine, since the compressed bitmap structure will be applied, the data structure will have the following three key parts:

1. a **Compressed B-Table** T, which is the main bitmap based structure, in which each row represented by a compressed bit vector–contains information about one transaction in the dataset;
2. an **HI-Counter** (Horizontal Index Counter) vector of level k (where k is the cardinality of the discovered patterns), which is an index counter list stored as a vector; and
3. a **VI-List** structure of level k. For each relevant domain item I in the current level k, the bit I is set to 1. Otherwise, it is set to 0. As we have observed from earlier sections, our adapted WAH-compressed bitmaps allow for efficient traversal of all bits set to 1. For all intents and purposes, this data structure

behaves like a list, but with the memory footprint of a compressed bit vector. This enables us to efficiently extract relevant information without the need to check if they belong in the current projected database.

Hence, each row in Compressed B-Table is a bit vector representing a transaction in the database. With the Compressed B-Table, we can keep the information of each transaction. A major benefit is that we can completely fill out this data structure on one scan of the database. Once we scan the database and fill out the data structure, we take each bit vector that comprises T and apply bitmap compression to it. Our compressed bit vectors consume less memory compared to regular, uncompressed ones in original B-mine and also eliminate data sparsity.

Alongside each compressed bit vector V, we store an index (called the fill map) as a bit vector B. The length of B is equivalent to the number of words in V, and the i-th bit of B is set to (a) 1 when the i-th word of V is a fill word and (b) 0 when it is a literal word. This index structure allows us to easily jump over literals and quickly identify fill words. We can use the CLZ and CTZ instructions to count the leading and/or trailing zeroes of the bit vector. Thus, we can also easily calculate where we are in the vector. By using this indexing structure, we can quickly access specific bit positions with internal functions.

Example 5.1 *Consider a compressed bit vector V with an index 0000 0000 0000 0000 0000 0000 0000 1000*$_{(16)}$*. This indicates that we have 32 words in our compressed vector. 31 of these are literal words, and 1 of them is a fill word in position 3 (indicated by the "1" in position 3 of the index). We can now calculate our position in the vector at each end of the fill word. We can use the CTZ instruction to count the trailing*

zeros in the index and see that we have skipped 3 literal words (each WAH word having 31 bits, thus we are at vector position 93). We can use an internal function to check the number of fill words contained in the fill at index position 3. If the fill consists of k fill words (again, each having 31 bits), then we know that we are at vector position $93 + 31k$ *(supposing the fill consists of 10 fill words, for instance, we would be at vector position 403).*

5.3 Summary

In this chapter, I described EB-mine, an enhanced B-mine algorithm with the ability of compressing bitmap data structure during the mining process. I made some significant modifications to make sure that the algorithm can perform data compression in an efficient way. In particular, I adapted and incorporated the word-aligned hybrid (WAH) compression scheme into my B-mine algorithm to form EB-mine. The evaluation of EB-mine will be presented in Chapter 7.

Chapter 6

Real-life Application: Social Network Mining with B-mine

Recall from Section 2.3, I briefly discussed the notion of "following" pattern in social network mining. In this chapter, I further introduce the notion of "following" patterns in social network mining in detail. After that, I introduce four frameworks, FoP-miner, ParFoP-miner, BigFoP and EFoP-miner, that apply B-mine, PB-mine, MRB-mine, and EB-mine to discover frequent "following" patterns in social networks.

6.1 The Social Network Analysis Problem of Finding "Following" Patterns

In this section, I first introduce the notion of "following" relationships (i.e., "followed" groups) in social networks. Then, I present how I can reduce the social network analysis problem of finding these "following" patterns into a data mining problem of

discovering collections of frequently followed social entities.

6.1.1 The "Following" Relationships

In social networking sites like Twitter or Weibo, social entities (users) are linked by "following" relationships (e.g., a user A follows another user B). Compared with the mutual friendship relationships in Facebook, this "following" relationship is different in that, the "following" relationships are directional. For instance, a user A may be following another user B while B is not following A. This property increases the complexity of the problem for two reasons. First, the group of users followed by A (e.g., A→{B, C, ...}) may not be same group of users as those who are following A (e.g., {..., X, Y, Z}→A). Due to asymmetry in the "following" relationships (cf. the symmetric friendship relationships), we cannot use the usual backtracking methods to determine the reverse relationships (e.g., we cannot determine whether or not E→A if we only know A→E). In this case, we need to check both directions to get the relationship between pairs of users. Second, in terms of computation, due to asymmetry in the "following" relationships, the computation time and space is doubled. For each pair of users, A→B (A follows B) does not automatically imply B→A. It requires double the amount of memory space to store the relationship between two users (i.e. storing two *directional* "following" relationships A→B and B→A), whereas one only needs to store one unidirectional friendship relationship A—B. This also means that, if there is a change in the dataset (e.g., A unfollows B), then we cannot remove the linkage between A and B because B may still be following A (cf. when A "unfriends" B, the friendship linkage between A and B disappears).

As the number of users in social networking sites (e.g., Twitter) is growing explosively, the number of relationships between social network users is also growing. One of the important research problems with regard to this high volume of data is to discover frequently "following" patterns.

A "following" pattern is a pattern representing the linkages when a significant number of users follow the same combination/group of users. For example, users who follow the twitter of NBA also follow the twitter of Adam Silver (current NBA commissioner). If there are large numbers of users who follow the twitters of both NBA and Adam Silver together, we can define this combination (NBA and Adam Silver) of followees as a frequent "following" pattern (i.e., a frequently followed group).

6.1.2 Example #1 of "Following" Relationships

For an illustrative purpose, consider a small portion of a social network represented as a graph shown in Fig. 6.1. Here, for N=6 users (Alice, Bob, Carol, Don, Ed, and Fred), (i) each node represents a user (i.e., a social entity) in the social network and (ii) the follow/subscribe relationships between pairs of users are represented by directed edges between the corresponding nodes. The arrow on an edge represents the "following" direction. For example, an unidirectional arrow "Bob→Carol" represents that Bob follows Carol (i.e., Bob follows Carol's page on a social networking site). Similarly, a bidirectional arrow "Alice↔Bob" represents that Alice and Bob are following each other (i.e., Alice follows Bob's page and Bob follows Carol's page on a social networking site). As seen in from the graph, each user is following some others as described below:

- Alice is following Bob and Ed.
- Bob is following Alice, Carol and Ed.
- Carol is following Alice and Ed.
- Don is following Bob, Carol and Ed.
- Ed is following Alice, Bob, Carol and Don.
- Fred is following Alice, Bob, Carol and Ed.

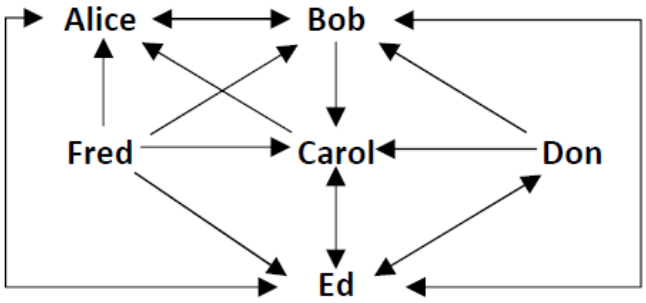

Figure 6.1: A sample social network graph containing six users. Directed edges between users represent the "following" relationship between users.

6.1.3 Example #2 of "Following" Relationships

Further, users in social networking sites are not confined to following pages of each other (e.g., portion of the social network shown in Fig. 6.1). Users can also follow a collection of M social networking pages belonging to different organizations (e.g., ACM, DEXA Society, IEEE) or on different sports. As another example, the above N=6 users may follow some of the M=6 social networking pages on different sports (e.g., **A**rchery, **B**aseball, **C**ycling, **D**iving, **E**questrian, **F**encing) as described below:

- Alice is following social networking pages B (on baseball) and E (on equestrian).
- Bob is following social networking pages A (on archery), C (on cycling) and E.
- Carol is following social networking pages A and E.
- Don is following social networking pages B, C and E.
- Ed is following social networking pages A, B, C and D (on diving).
- Fred is following social networking pages A, B, C and E.

6.1.4 Reduction to a Frequent Pattern Mining Problem

Given a social network of N social entities, each of them is *following* some other social entities, for a total of M distinct pages for individual users and/or organizations. A list of these pages can be captured "horizontally" for each user. Equivalently, a list of followers can be "vertically" associated with each of these M distinct pages for individual users and/or organizations. An important research problem in social network analysis is to mine interesting "following" patterns (i.e., collections of social network pages that are frequently followed by users).

Recall from Chapter 1 that the research problem of frequent pattern mining aims to find sets of all frequently co-occurring items from a shopper market basket database consisting of N transactions. Each of these transactions can "horizontally" capture a set of items (e.g., co-located events, merchandise items in the same shopper market basket), for a total of M distinct domain items in the database. Equivalently, for each of the M domain items, a list of transactions containing such an item can be captured "vertically".

Observing the similarities between the two problems, we can reduce the social network analysis problem of mining interesting "following" patterns into a data mining problem of finding popular collections of social networking pages that are frequently followed by social entities. Note that, although I present the proposed work as a social network analysis problem of mining interesting "following" patterns, we are not confined to this domain. The idea can be generalized to a data mining problem of finding popular collections of social networking pages that are frequently followed by social entities. To a further extent, it can be generalized to a data mining problem of finding frequent patterns (i.e., sets of frequently co-occurring items) from classical transaction databases.

6.2 The FoP-miner Framework

For this framework, I am applying B-mine algorithm to mine "following" patterns. Thus, the data structure and mining process are very similar to the original B-mine algorithm.

6.2.1 The FoP-structure

To discover frequent "following" relationships in social networks, we propose a new data structure—called **FoP-structure**—to capture important social information (especially, "following" relationships among social entities) for mining "**Fo**llowing" **P**atterns from social networks. Specifically, for each user p in a social network, we capture a list of social networking pages followed by p. We capture N such lists, for a total of N users in the network. We represent this information in a bitwise tabular

form as was done in the original B-mine algorithm. Such a table forms the basis of our data structure, which contains the following three parts:

1. a **SocialTable**, which is the main bitmap-based structure, in which each row contains information about one particular user (a follower and an associated list of its followees or followed pages) in a social network;

2. a **Horizontal Index Counter (HI-Counter)** structure, which is an array (or vector) of column sums counting the number of followers for each social networking page; and

3. a **Vertical Index List (VI-List)** structure, which is a two-dimensional data structure in which each row contains (a) a prefix of the current row in the VI-List structure indicating a group of followees and (b) its corresponding list of followers based on the location of the first relevant "1" bit.

To capture the important contents of a social network (i.e. lists of social networking pages followed by users), I construct the SocialTable as well as its associated HI-Counter and VI-List structures as follows:

1. For each list L_j of social networking pages followed by a user p, we create a bitmap row of size M, where M is the total number of distinct social networking pages in the social network. Each column (represented by a bit) in the row represents the social networking page in each list L_j. In other words, we put a "1" in the i-th bit (where $1 \leq i \leq M$) if the i-th social networking page is present in L_j for a user p; we put a "0" otherwise (i.e., when the i-th social

networking page is absent from L_j). When we repeat the aforementioned actions in this step, we get multiple bitmap rows to form a SocialTable.

2. For each column c of the SocialTable, we compute a top-level HI-Counter structure. Specifically, we count the number of 1s in column c and put the count (i.e., column sum) in the c-th position/entry of the HI-Counter structure.

3. For each row r of the SocialTable, we create a top-level VI-List structure. Specifically, we locate the occurrence of the first "1" bit in row r and group rows sharing the same column where the occurrence of the first "1" bit is. Hence, for M columns in the SocialTable, there are at most M groups in this VI-List structure.

Note that all the above steps can be completed in only one scan of the social network database.

6.2.2 Example of the FoP-structure

The social network described in Sections 6.1.2 and 6.1.3 can be represented by a FoP-structure, as shown in Table 6.1. An entry of "1" in (row X, column Y) in the SocialTable indicates the presence of a link from X to Y. In the social network context, this means that X follows Y. An entry of "0" in (row X, column Y) indicates the absence of a link from X to Y. As the "follow" action is not symmetric, X follows Y does not guarantee that Y follows X. For example, Fred follows Alice's social networking page A as indicated by the presence of "1" in (row Fred, column A), but Alice does not follow Fred's social networking page F as indicated by the presence of

"0" in (row Alice, column F). Here, as Alice is following social networking pages B and E, we (i) put 1s in both columns B & E and (ii) put 0s in the remaining columns (i.e., columns A, C, D and F) in the SocialTable.

Table 6.1: The FoP-structure (i.e., SocialTable and its associated top-level HI-Counter & VI-List structures)

SocialTable:	Followees					
Follower	A	B	C	D	E	F
Alice	0	**1**	0	0	1	0
Bob	**1**	0	1	0	1	0
Carol	**1**	0	0	0	1	0
Don	0	**1**	1	0	1	0
Ed	**1**	1	1	1	0	0
Fred	**1**	1	1	0	1	0

HI-Counter:	Followees					
	A	B	C	D	E	F
Colum sum:	**4**	**4**	4	1	5	0

VI-List:	
Followee	List of followers
{A}	Bob, Carol, Ed, Fred
{B}	Alice, Don

For each column c of the SocialTable, we compute a column sum by counting the number of 1s in column c. For instance, the social networking page A is followed by Bob, Carol, Ed and Fred as indicated by the four 1s in column A. So, the column sum for column A in the HI-Counter structure shown in Table 6.1 becomes 4. Each sum is at most 6 (because there are only N=6 users/followers).

For the row for Alice, the first occurrence of a "1" bit is in column B; For the row for Don, the first occurrence of a "1" bit is also in column B. Hence, the VI-List structure shown in Table 6.1 contains information that the social networking page B

that is followed by Alice & Don. Similarly, the first occurrences of a "1" bit in rows for Bob, Carol, Ed and Fred are all in column A. Hence, the VI-List structure also contains information that the social networking page A is followed by Bob, Carol, Ed & Fred.

From this example, we observe the following general properties about the FoP-structure consisting of a SocialTable, HI-Counter and VI-List structures:

- Only one scan of the social network data is required to build our *SocialTable* (cf. those frequent pattern mining algorithms, such as the hyperlink based algorithm H-mine [PHL+01], requiring two scans of data). Moreover, the SocialTable does not contain a high number of hyperlinks. Furthermore, each cell/entry in the SocialTable takes only 1 bit in memory (cf. pointers in H-mine).

- The *HI-Counter* structure is an array/vector of non-negative integers (ranging from 0 to N), where N is the total number of users/followers. The size/length of the HI-Counter structure is M, which is the total number of distinct social networking pages/followees.

- The number of groups/lists of followees in the *VI-List* structure may vary depending on the grouping (i.e., the location of the relevant first 1 bit). However, the number of groups/lists is bounded above by M, which is the total number of distinct social networking pages/followees. Moreover, the total number of followers among all the groups/lists is bounded above by N, which is the total number of users/followers.

6.2.3 The Mining Process in FoP-miner

Once the FoP-structure (consisting of the SocialTable and its associated top-level HI-Counter & VI-List structures) are constructed, my social mining algorithm—called **FoP-miner**—can discover interesting "following" patterns from the relevant portion of the social network captured by this FoP-structure.

The key idea of my social network mining algorithm can be described as follows. The algorithm finds individual frequently followed social networking pages by checking column sums in the HI-Counter structure. Followees with column sums $\geq$ a user-specified frequency threshold are considered as frequent. Based on the contents of the VI-List structure, non-singleton groups (e.g., pairs, triplets, quadruplets) of frequently followed social networking pages can then be discovered. Specifically, we consider only rows belonging to the same group in the VI-List structure for frequent $\{x\}$ (i.e. the column sum of $\{x\} \geq$ the user-specified frequency threshold) as the projected SocialTable for $\{x\}$. The HI-Counter structure for the $\{x\}$-projected SocialTable can then be computed. Again, if the column sum for any followee w in this new HI-Counter structure for the $\{x\}$-projected SocialTable $\geq$ the user-specified frequency threshold, then $\{w, x\}$ is considered a pair of frequently followed social networking pages, from which a new VI-List structure for the projected SocialTable for $\{w, x\}$ can then be formed. This mining process is repeated until no more VI-List structures can be formed (or need to be formed), meaning all relevant lists of social networking pages have been examined and all interesting "following" patterns (i.e., interesting groups of frequently followed social networking pages) have been discovered from the relevant portion of the social network.

To illustrate my serial social network mining process, let us continue with the FoP-structure (shown in Table 6.1) constructed in Section 6.2.2. Let the user-specified frequency threshold be 2, meaning that groups/lists of social networking pages in the resulting "following" patterns must be followed by at least 2 social entities/followers to be considered *frequent* or *interesting*.

With this setting, columns D and F can be ignored because their column sums in the top-level HI-Counter (as shown in Table 6.1) are below the user-specified frequency threshold. On the other hand, based on the column sums in this top-level HI-Counter structure, we discover that social networking page A is followed by 4 users (namely, Bob, Carol, Ed and Fred). So, {social networking page A} is considered frequent or interesting. Similarly, social networking page B is also followed by 4 users (this time, Alice, Don, Ed and Fred). Social networking page C is followed by 4 users (Bob, Don, Ed and Fred), and social networking page E is followed by 5 users (Alice, Bob, Carol, Don and Fred). In other words, our mining algorithm found four interesting singleton "following" patterns {social networking page A}, {social networking page B}, {social networking page C} and {social networking page E}.

We then use the top-level VI-List structure (as shown in Table 6.1) to extract all relevant rows/followers (i.e., rows for Bob, Carol, Ed and Fred) from the SocialTable to compute the projected SocialTable for followee {A}, from which we construct its corresponding HI-Counter and VI-List structures as shown in Table 6.2. *It is important to note that the contents of the SocialTable remain unchanged throughout the entire mining process.* We only construct new HI-Counter and VI-List structures. For readability, we highlight (by boldfacing) those relevant rows and columns in the

SocialTable.

Table 6.2: A FoP-structure for {social networking page A}

SocialTable:	Followees					
Follower	A	**B**	**C**	D	**E**	F
Alice	0	1	0	0	1	0
Bob	1	**0**	**1**	0	**1**	0
Carol	1	**0**	**0**	0	**1**	0
Don	0	1	1	0	1	0
Ed	1	**1**	**1**	1	**0**	0
Fred	1	**1**	**1**	0	**1**	0
HI-Counter:		**2**	**3**		**3**	

VI-List:	
Followee	List of followers
{A, B}	Ed, Fred
{A, C}	Bob
{A, E}	Carol

Based on the new HI-Counter structure for {social networking page A} (as shown in Table 6.2), we discover that social networking pages A & B are followed together by 2 users (namely, Ed and Fred). So, {social networking pages A, B} is considered *frequent* or *interesting*. Similarly, social networking pages A & C are followed together by 3 users (Bob, Ed and Fred), and social networking pages A & E are followed together by 3 users (Bob, Carol and Fred). In other words, our mining algorithm found three interesting "following" patterns in terms of pairs of frequently followed pages: {social networking pages A, B}, {social networking pages A, C} and {social networking pages A, E}.

We then use the new VI-List structure for {social networking page A} (as shown in Table 6.2) to extract all relevant rows/followers (i.e., rows for Ed and Fred) to compute the projected SocialTable for followee {A, B}, from which we construct its

corresponding HI-Counter structure as shown in Table 6.3. Again, those relevant rows and columns are highlighted.

Table 6.3: The FoP-structure for {social networking pages A, B}

SocialTable:	Followees					
Follower	A	B	**C**	D	**E**	F
Alice	0	1	0	0	1	0
Bob	1	0	1	0	1	0
Carol	1	0	0	0	1	0
Don	0	1	1	0	1	0
Ed	1	1	**1**	1	**0**	0
Fred	1	1	**1**	0	**1**	0
HI-Counter:			**2**		**1**	

Based on this new HI-Counter structure for {social networking pages A, B} (as shown in Table 6.3), we discover that social networking pages A, B & C are followed together by 2 users (namely, Ed and Fred). So, {social networking pages A, B, C} is considered an interesting "following" pattern.

However, the column sum for E in the HI-Counter structure in Table 6.3 is below the user-specified frequency threshold, and column E can thus be ignored. This also means that {social networking pages A, B, E} is infrequent/uninteresting.

Note that we do not need to construct a VI-List structure for {social networking pages A, B} because, after ignoring column E, there would be only one followee group/list (namely, followee {A, B, C}) with 2 followers (i.e., Ed & Fred in the corresponding list of followers). No additional social networking page would be *frequently* followed together with pages A, B & C by these 2 followers. (To double check, an observant reader can note that {social networking pages A, B, C, E} is only followed by Fred and is thus infrequent/uninteresting. Even without counting, one would ex-

pect that {social networking pages A, B, C, E} is infrequent/uninteresting as {social networking pages A, B, E} is infrequent/uninteresting.) Hence, we backtrack instead, and update the VI-List structure for {social networking page A}. See Table 6.4 for an updated FoP-structure (especially, the updated VI-List structures).

Table 6.4: An updated FoP-structure for {social networking page A} after mining {pages A, B}

SocialTable:	Followees					
Follower	A	B	**C**	D	**E**	F
Alice	0	1	0	0	1	0
Bob	1	0	**1**	0	**1**	0
Carol	1	0	**0**	0	**1**	0
Don	0	1	1	0	1	0
Ed	1	1	**1**	1	**0**	0
Fred	1	1	**1**	0	**1**	0
HI-Counter:			**3**		**3**	

VI-List:	
Followee	List of followers
{A, C}	Bob, **Ed, Fred**
{A, E}	Carol

Note that the HI-Counter structure for {social networking page A} (as shown in Table 6.4) is updated to contain only column sums of followees C and E (both with frequency value 3) after mining {pages A, B}. Moreover, by using the updated VI-List structure for {social networking page A}, we extract all relevant rows/followers (i.e., rows for Bob, Ed and Fred) to compute the projected SocialTable for followee {A, C}, from which we construct its corresponding HI-Counter structure as shown in Table 6.5.

Based on this new HI-Counter structure for {social networking pages A, C} (as shown in Table 6.5), we discover that social networking pages A, C & E are fol-

Table 6.5: The FoP-structure for {social networking pages A, C}

SocialTable:	Followees					
Follower	A	B	C	D	**E**	F
Alice	0	1	0	0	1	0
Bob	1	0	1	0	**1**	0
Carol	1	0	0	0	1	0
Don	0	1	1	0	1	0
Ed	1	1	1	1	**0**	0
Fred	1	1	1	0	**1**	0
HI-Counter:					**2**	

lowed together by 2 users (Bob and Fred). So, {social networking pages A, C, E} is considered an interesting "following" pattern.

Again, we do not need to construct a VI-List structure for {social networking pages A, C}, but for a slightly different reason. Specifically, there would be only one followee group/list (namely, followee {A, C, E}) with 2 followers (i.e., Bob & Fred in the corresponding list of followers), but no additional social networking page would be followed together with pages A, C & E by these 2 followers. Hence, we backtrack instead.

Note that, when backtracking, it is tempting to update the VI-List structure for {social networking page A} a second time to get (i) an updated HI-Counter structure containing only the column sum of followee E (with frequency value 3) after mining {pages A, C} and (ii) an updated VI-List structure containing only one followee group/list (namely, followee {A, E}) with 3 followers (i.e., Bob, Carol & Fred in the corresponding list of followers). However, no additional social networking page would be *frequently* followed together with pages A & E by these 3 followers. Hence, when backtracking, we update the top-level VI-List structure. See Table 6.6 for an updated

FoP-structure (especially, an updated VI-List structure).

Table 6.6: An updated FoP-structure for after mining the pages frequently followed together with {page A}

SocialTable:	Followees					
Follower	A	**B**	**C**	D	**E**	F
Alice	0	**1**	0	0	1	0
Bob	1	0	**1**	0	1	0
Carol	1	0	0	0	**1**	0
Don	0	**1**	1	0	1	0
Ed	1	**1**	1	1	0	0
Fred	1	**1**	1	0	1	0
HI-Counter:		**4**	**4**		**5**	

VI-List:	
Followee	List of followers
{B}	Alice, Don, **Ed, Fred**
{C}	**Bob**
{E}	**Carol**

At this point, we have found all interesting "following" patterns containing social networking page A. We apply similar steps to find all interesting "following" patterns containing other social networking pages (e.g., pages B, C, E): {social networking pages B, C}, {social networking pages B, E}, and {social networking pages B, C, E}.

After mining the social networking pages that are frequently followed together with {page B}, we once again update the top-level FoP-structure. See Table 6.7 for this updated FoP-structure (especially, an updated VI-List structure). With this updated FoP-structure, we find another interesting "following" pattern {social networking pages C, E}.

Note that, after mining the social networking pages frequently followed together with {page C}, we do not need to update the top-level FoP-structure because (i) there

Table 6.7: An updated FoP-structure for after mining the pages frequently followed together with {page B}

SocialTable:	Followees					
Follower	A	B	**C**	D	**E**	F
Alice	0	1	0	0	**1**	0
Bob	1	0	**1**	0	1	0
Carol	1	0	0	0	**1**	0
Don	0	1	**1**	0	1	0
Ed	1	1	**1**	1	0	0
Fred	1	1	**1**	0	1	0
HI-Counter:			**4**		**5**	

VI-List:	
Followee	List of followers
{C}	Bob, **Don, Ed, Fred**
{E}	**Alice**, Carol

would be only one followee group/list (namely, followee {E}) with 5 followers (i.e., Alice, Bob, Carol, Don & Fred in the corresponding list of followers), but no additional social networking page would be followed together with page E by these 5 followers. At this time, we have found all 13 interesting "following" patterns from social networks in Section 6.1.2 or 6.1.3.

From this example, we observe the following general properties about the FoP-miner algorithm:

- During the mining process, the contents of the SocialTable remain unchanged. The algorithm just examines relevant rows and columns in the SocialTable. The list of followers in the VI-List structure for a followee (e.g., {A,B} in Table 6.2) indicates the relevant rows/followers to be examined (e.g., rows for Ed & Fred in the corresponding FoP-structure for {social networking pages A, B} shown in Table 6.3).

- The number of groups/lists of followees in the VI-List structure is bounded above by M, which is the total number of frequent/interesting followees in the HI-Counter structure (e.g., 2 lists of followers for the top-level HI-Counter structure containing 4 frequent followees/column sums in Table 6.1, 3 lists of followers for the HI-Counter structure for {social networking page A} containing 3 frequent followees/column sums in Table 6.2).

- When the number of frequent/interesting followees in the HI-Counter structure drops to 1, the algorithm does not need to construct a VI-List structure. For example, when the HI-Counter structure for {social networking page A, C} containing only 1 frequent followee/column sum E in Table 6.5, the algorithm does not construct any VI-List structure because no additional social networking page could be followed together with pages A, C & E. As another example, when the HI-Counter structure for {social networking page A, B} contains only 1 frequent followee/column sum C (E is infrequent) in Table 6.3, the algorithm does not construct any VI-List structure because no additional social networking page would be frequently followed together with pages A, B & C.

6.3 The ParFoP-miner Framework

The ParFoP-miner Framework, as its name suggests, applies parallel computing. Recall from Section 4.1, PB-mine (the parallel version of B-mine) can distribute mining jobs to different processors because of the data independency of the B-table in the original B-mine. In this section, we see how this approach can be adapted to

my social network mining problem using an example social network.

The key idea of my ParFoP-miner framework can be described as follows. Based on the top-level SocialTable, the algorithm extracts from the social network the information relevant to each followee (or social networking page) and passes the information to available processors. Each processor then mines interesting "following" patterns from this information in parallel. To make this happen, the algorithm extracts the information from the social network in such a way that the mining can be performed independently (i.e., one processor does not need to wait for results from another processor to start the mining). Specifically, based on the top-level SocialTable, each processor computes an $\{x\}$-projected SocialTable, which contains all followers of followee $\{x\}$, for every frequent/interesting followee $\{x\}$. From the $\{x\}$-projected SocialTable, interesting "following" patterns are then mined in a similar fashion as that by the serial counterpart (i.e., FoP-miner algorithm). As a preview, my ParFoP-miner extracts from the social network and passes (i) the information relevant to followee {A} (i.e., rows about followers Bob, Carol, Ed and Fred) to processor 1, (ii) the information relevant to followee {B} (i.e., rows about followers Alice, Don, Ed and Fred) to processor 2, and (iii) the information relevant to followee {C} (i.e., rows about followers Bob, Don, Ed and Fred) to processor 3.

Section 6.3.1 shows a FoP-structure with this ParVI-List structure for the parallel social network mining problem, and Section 6.3.2 illustrates how my parallel social network mining framework discovers all interesting "following" patterns from social networks.

6.3.1 Example of the FoP-structure for Parallel Social Network Mining

The social network example described in Sections 6.1.2 and 6.1.3 can be represented by a FoP-structure, as shown in Table 6.8. While the SocialTable and HI-Counter structure are identical to that shown in Table 6.1, the ParVI-List structure here is different from the VI-List structure in Table 6.1. Recall that the VI-List structure for serial mining was constructed based on the first occurrences of a "1" bit in each row. So, given that the first occurrences of a "1" bit in rows for Alice & Don are in column B, the VI-List structure contains information that the social networking page B is followed Alice & Don. Such a row in the VI-List structure is updated after mining those pages frequently followed together with {A} to become that shown in Table 6.6. This table showed that (i) {B} is followed by Alice, Don, Ed & Fred, and (ii) {C} is followed by Bob. The latter is again updated after mining those pages frequently followed together with {B} to become that shown in Table 6.7. It showed that {C} is followed by Bob, Don, Ed & Fred.

Note that all this information is captured by ParVI-List structure without any need of updates. The reason is that the ParVI-List structure for parallel mining is constructed based on *all* occurrences (instead of just the *first* occurrence) of a "1" bit in each row.

Table 6.8: The FoP-structure (i.e., SocialTable and its associated top-level HI-Counter & ParVI-List structures)

SocialTable:	Followees					
Follower	A	B	C	D	E	F
Alice	0	1	0	0	1	0
Bob	1	0	1	0	1	0
Carol	1	0	0	0	1	0
Don	0	1	1	0	1	0
Ed	1	1	1	1	0	0
Fred	1	1	1	0	1	0
HI-Counter:	4	4	4	1	5	0

ParVI-List:	
Followee	List of followers
{A}	Bob, Carol, Ed, Fred
{B}	Alice, Don, Ed, Fred
{C}	Bob, Don, Ed, Fred
{E}	Alice, Bob, Carol, Don, Fred

6.3.2 Example of the Parallel Mining Algorithm ParFoP-miner

To illustrate my parallel social network mining process, let us continue with the FoP-structure (shown in Table 6.8). Let the user-specified frequency threshold be 2, meaning that groups/lists of social networking pages in the resulting "following" patterns must be followed by at least 2 social entities/followers in order to be considered *frequent* or *interesting*. With this setting, columns D and F can be ignored as their column sums in the top-level HI-Counter structure are below the user-specified frequency threshold. On the other hand, based on the column sums in this top-level HI-Counter structure, my mining algorithm found four interesting singleton "following" patterns {social networking page A}, {social networking page B}, {social networking page C}

and {social networking page E}.

Processors 1, 2 and 3 then use the top-level ParVI-List structure to extract all relevant rows/followers from the SocialTable to compute projected SocialTables for followees {A}, {B} and {C}, from which their corresponding HI-Counter and VI-List structures are constructed as shown in Table 6.9 and Table 6.10. Note that the ParVI-List structure is only constructed for the top-level. In all subsequent levels, the usual VI-List structures can be used unless the projected SocialTable for such a level is further parallelized (say, for load balancing). Moreover, each processor only considers those relevant rows and columns, where relevant rows are indicated by the top-level ParVI-List structure. For instance, Processor 1, which computes the projected SocialTable for followee {A}, considers only columns B, C & E (as D & F are infrequent/uninteresting) in rows for followers Bob, Carol, Ed and Fred. Similarly, Processor 2, which computes the projected SocialTable for followee {B}, considers only columns C & E in rows for followers Alice, Don, Ed and Fred. Processor 3, which computes the projected SocialTable for followee {C}, considers only column E in rows for followers Bob, Don, Ed and Fred. Note that no processor is needed to compute the projected SocialTable for followee {E} because no additional social networking page could be frequently followed together with page E.

Once each processor computes its SocialTable, the corresponding HI-Counter and VI-List structures can be easily constructed in the same way as FoP-miner. Based on these new HI-Counter structures, Processor 1 finds three interesting "following" patterns in terms of pairs of frequently followed pages (namely, {social networking pages A, B}, {social networking pages A, C} and {social networking pages A, E}).

Table 6.9: SocialTables and HI-Counter structures for domain items {A}, {B} and {C}

SocialTable for {A}:	Followees		
Follower	B	C	E
Bob	0	1	1
Carol	0	0	1
Ed	1	1	0
Fred	1	1	1
HI-Counter for {A}:	2	3	3

SocialTable for {B}:	Followees	
Follower	C	E
Alice	0	1
Don	1	1
Ed	1	0
Fred	1	1
HI-Counter for {B}:	3	3

SocialTable for {C}:	Followee
Follower	E
Bob	1
Don	1
Ed	0
Fred	1
HI-Counter for {C}:	3

Processor 2 finds two interesting "following" patterns (namely, {social networking pages B, C} and {social networking pages B, E}); Processor 3 finds an interesting "following" pattern (namely, {social networking pages C, E}). Based on the VI-List structures, Processor 1 computes SocialTable for {A, B} to find an interesting "following" pattern {social networking pages A, B, C}. Afterwards, Processor 1 updates the VI-List structure for {A} by adding Ed & Fred as followers to {A, C}

Table 6.10: VI-List structures for domain items {A}, {B} and {C}

VI-List for {A}:

Followee	Followers
{A, B}	Ed, Fred
{A, C}	Bob
{A, E}	Carol

VI-List for {B}:

Followee	Followers
{B, C}	Don, Ed, Fred
{B, E}	Alice

VI-List for {C}:

Followee	Followers
{C, E}	Bob, Don, Fred

and computes SocialTable for {A, C} to find interesting "following" pattern {social networking pages A, C, E}. Similarly, Processor 2 computes SocialTable for {B, C} to find {social networking pages B, C, E} as the 13th interesting "following" pattern.

Note that, in the above illustrative example, our ParFoP-miner used 3 processors (for followees/social networking pages {A}, {B} and {C}) because there are only 3+1 = 4 frequent/interesting followees. With 4 frequent/interesting followees ({A}, {B}, {C} and {E}), we do not need a processor for the last followee (e.g., {E}) as no additional followee could be frequently followed together with {E}. In general, for K frequent/interesting followees, ParFoP-miner requires only $K-1$ processors. If there are surplus of available processors, ParFoP-miner can further parallelize some projected SocialTable to take advantages of extra processors. On the other hand, if there is a shortage of available processors, ParFoP-miner can merge relevant rows/followers for multiple followees/social networking pages.

6.4 The BigFoP Framework

FoP-miner is a framework that applies the MRB-mine algorithm to mining "following" patterns from a social network. The MRB-mine algorithm has been introduced in Section 4.2. In this section, I will walk through this framework by using the following social network dataset as shown in Figure 6.1.

As expected, the first step is to convert Figure 6.1 in to a bit-map representation as shown in Table 6.11, that the MRB-mine can be applied to.

Table 6.11: The B-table Constructed from Figure 6.1

Follower	Followees					
	Alice	Bob	Carol	Don	Ed	Fred
Alice	0	**1**	0	0	1	0
Bob	**1**	0	1	0	1	0
Carol	**1**	0	0	0	1	0
Don	0	**1**	1	0	1	0
Ed	**1**	1	1	1	0	0
Fred	**1**	1	1	0	1	0

6.4.1 The First Set of Map-Reduce Functions in BigFoP

Abstractly, BigFoP first applies a map function to each edge as follows:

$$\begin{aligned} \text{map}_1\text{:}\ & \langle \text{edge ID, "following" relationship captured by the edge} \rangle \\ & \mapsto \langle \text{follower, individual followee} \rangle, \end{aligned} \tag{6.1}$$

in which the master node reads and divides the social network data into partitions. Specifically, the map_1 function can be specified as follows:

for each edge e=⟨follower, followee⟩ $\in E$ in social network G=(V, E) **do**

emit ⟨follower, followee, 1⟩.

This map function is applied to each edge e=⟨follower, followee⟩ $\in E$ in the social network represented by $G = (V, E)$, and results in a list of ⟨follower, followee, 1⟩ capturing all existing "following" relationships (between followers and followees) in the social network. See Example 6.1.

Example 6.1 *After applying the* map_1 *function to the example social network dataset in Figure 6.1, the BigFoP framework returns a list containing* ⟨*Alice, B, 1*⟩, ⟨*Alice, E, 1*⟩, ⟨*Bob, A, 1*⟩, ⟨*Bob, C, 1*⟩, ⟨*Bob, E, 1*⟩, ⟨*Carol, A, 1*⟩, ⟨*Carol, E, 1*⟩, ⟨*Don, B, 1*⟩, ⟨*Don, C, 1*⟩, ⟨*Don, E, 1*⟩, ⟨*Ed, A, 1*⟩, ⟨*Ed, B, 1*⟩, ⟨*Ed, C, 1*⟩, ⟨*Ed, D, 1*⟩, ⟨*Fred, A, 1*⟩, ⟨*Fred, B, 1*⟩, ⟨*Fred, C, 1*⟩, *and* ⟨*Fred, E, 1*⟩. ■

Afterwards, BigFoP applies a reduce function to group and count the number of followers for each followee, as well as to list these followers for each followee. More specifically, ⟨follower, followee, 1⟩ pairs from the map_1 function are shuffled and sorted. Each processor then executes the reduce function on the shuffled and sorted pairs to count the number of followers and list them for each followee. To speed up this big social network data mining process, BigFoP also allows users to specify the *interestingness* of groups of social entities by a frequency threshold. Here, the users can indicate the minimum number of followers for a group of followees so that the group can be considered interesting. By incorporating this user preference, BigFoP returns (i) a list of followers only for those popular followees (i.e., followees who are frequently followed by at least the minimum number of followers) and (ii) the count

for each followee. In other words, BigFoP applies the following reduce function:

$$\begin{aligned} \text{reduce}_1\text{:}\ & \langle \text{followee, list of followers} \rangle \\ & \mapsto \text{list of } \langle \textit{interesting} \text{ followee, follower information} \rangle, \end{aligned} \tag{6.2}$$

with a detailed definition as follows:

for each followee $\in \langle \_, \text{followee}, \_ \rangle$ emitted by map_1 **do**
  set counter[followee] = 0;
  set list[followee] = {};
  for each follower $\in \langle \text{follower, followee}, 1 \rangle$ emitted by map_1 **do**
    counter[followee] = counter[followee] + 1;
    list[followee] = list[followee] $\cup$ {follower};
  if counter[followee] $\geq$ user-specified min frequency threshold
  then emit ⟨followee, counter[followee], list[followee]⟩.

This results in (i) a list of followers and (ii) its count for each *interesting/popular* followee. See Example 6.2.

Example 6.2 *Continue with Example 6.1. BigFoP applies the* reduce_1 *function with user-specified minimum frequency threshold of 2 followers and returns* ⟨*A, 4,* {*Bob, Carol, Ed, Fred*}⟩, ⟨*B, 4,* {*Alice, Don, Ed, Fred*}⟩, ⟨*C, 4,* {*Bob, Don, Ed, Fred*}⟩, *and* ⟨*E, 5,* {*Alice, Bob, Carol, Don, Fred*}⟩. *Note that BigFoP does not return the lists for followees D or F because their corresponding counters were low (D and F were followed by only 1 and 0 followers, respectively).*

To summarize, after applying the first set of map_1 *and* reduce_1 *functions, BigFoP*

has so far discovered four interesting "following" patterns—in the form of individual *frequently followed social entities—namely,* $\{A\}$*,* $\{B\}$*,* $\{C\}$ *and* $\{E\}$*, who are followed by 4, 4, 4 and 5 followers respectively. Each of these four individual followees is, naturally, followed by at least 2 followers (the user-specified minimum frequency threshold).* ■

6.4.2 The Second Set of Map-Reduce Functions in BigFoP

Thereafter, BigFoP applies the next set of map and reduce functions to mine interesting "following" patterns in the form of *pairs* of frequently followed social entities based on the results from the first set of map_1 and reduce_1 functions. For instance, knowing that D and E are unpopular individual followees, it is guaranteed that any pairs containing followee D or E is also unpopular. By making use of this knowledge, the search space for mining interesting "following" patterns can then be pruned effectively. Specifically, the map_2 function, which returns $\langle$follower, $\{p\} \cup \{\text{followee}\}$, $1\rangle$ for every follower in the follower list of each popular/interesting individual followee p, can be specified as follows:

$$\begin{aligned} \text{map}_2\text{:}\ & \langle\text{interesting followee } p\text{, its follower information}\rangle \\ & \mapsto \langle\text{follower, followee pair}\rangle, \end{aligned} \tag{6.3}$$

with a detailed definition as follows:

for each $p \in \langle p, \_, \text{list}[p]\rangle$ emitted by reduce_1 **do**

  for each follower $\in \text{list}[p]$ **do**

for each ⟨follower, followee⟩ $\in E$ of social network G=(V, E) **do**

if isRelevant(followee, p)

then emit ⟨follower, $\{p\} \cup \{$followee$\}$, 1⟩.

Here, isRelevant(followee, p) is a Boolean function checking the relevance (e.g., consistence to the mining order) of followee with respect to p. This results in lists of ⟨follower, followee, 1⟩, and a list for each popular individual followee p returned by the reduce$_1$ function. See Example 6.3.

Example 6.3 *Continue with Example 6.2. Recall that the first set of map$_1$ and reduce$_1$ functions returns four popular followees A, B, C and E. So, for popular followee A (followed by four followers Bob, Carol, Ed & Fred), the map$_2$ function emits all* relevant *followees of these four followers: ⟨Bob, AC, 1⟩, ⟨Bob, AE, 1⟩, ⟨Carol, AE, 1⟩, ⟨Ed, AB, 1⟩, ⟨Ed, AC, 1⟩, ⟨Fred, AB, 1⟩, ⟨Fred, AC, 1⟩, and ⟨Fred, AE, 1⟩. Note that (i) followees of Alice are not emitted (because it is not meaningful for Alice to follow himself), (ii) followees of Don are not emitted (because Don does not follow A), (iii) four relationships of the form ⟨_, A, 1⟩ (e.g., ⟨Bob, A, 1⟩) are irrelevant with respect to p=A (because we already knew these four followers are following* single individual followee *A when we started this map$_2$ function & we aimed to find followers who follow* pairs of followees*), and (iv) ⟨Ed, AD, 1⟩ is also irrelevant (because followee D is unpopular).*

Similarly, for popular followee B (followed by four followers Alice, Don, Ed & Fred), the map$_2$ function emits all relevant *followees of these four followers: {⟨Alice, BE, 1⟩, ⟨Don, BC, 1⟩, ⟨Don, BE, 1⟩, ⟨Ed, BC, 1⟩, ⟨Fred, BC, 1⟩, ⟨Fred, BE, 1⟩}. Note that (i) followees of Bob are not emitted (because it is not meaningful for Bob to*

follow himself), (ii) followees of Carol are not emitted (because Carol does not follow B), (iii) four relationships of the form $\langle \_, B, 1\rangle$ *(e.g.,* $\langle Don, B, 1\rangle$*) are irrelevant with respect to p=B (because we already knew these four followers are following* single individual followee *B when we started this* map_2 *function and we aimed to find followers who follow* pairs of followees*), and (iv)* $\langle Ed, BD, 1\rangle$ *is also irrelevant (because followee D is unpopular). More important to note is that (v) relationships in the form* $\langle \_, AB, 1\rangle$ *(e.g.,* $\langle Ed, AB, 1\rangle$*,* $\langle Fred, AB, 1\rangle$*) are irrelevant with respect to p=B (because these relationships are already processed by the* map_2 *function).*

Then, for popular followee C (followed by four followers Bob, Don, Ed & Fred), the map_2 *function emits all* relevant *followees of these four followers:* $\{\langle Bob, CE, 1\rangle$, $\langle Don, CE, 1\rangle$, $\langle Fred, CE, 1\rangle\}$.

Finally, for popular followee E (followed by five followers Alice, Bob, Carol, Don & Fred), the map_2 *function does not emit any followee because there is no* relevant *followee for these five followers.* ■

Similar to reduce_1, the reduce_2 function shuffles and sorts $\langle$follower, $\{p\} \cup \{$relevant followee$\}$, $1\rangle$ to find and count followers for each followee pair $P = (\{p\} \cup \{$relevant followee$\})$ as follows:

$$\begin{aligned} \text{reduce}_2\text{:}\ & \langle\text{followee pair, list of common followers}\rangle \\ & \mapsto \text{list of } \langle \textit{interesting} \text{ followee pair, follower information}\rangle, \end{aligned} \tag{6.4}$$

with a detailed definition as follows:

for each $P \in \langle \_,$ followee group $P, \_\rangle$ emitted by map_2 **do**

set counter[P] = 0;

set list[P] = {};

for each follower $\in \langle$follower, P, 1$\rangle$ emitted by map$_2$ **do**

counter[P] = counter[P] + 1;

list[P] = list[P] $\cup$ {follower};

if counter[P] $\geq$ user-specified min frequency threshold

then emit $\langle P$, counter[P], list[P]$\rangle$.

This results in (i) a list of followers and (ii) its count for each *interesting/popular* followee pair P.

Example 6.4 *Continue with Example 6.3. BigFoP applies the reduce$_2$ function with user-specified minimum frequency threshold = 2 followers and returns* $\langle$*AB, 2,* {*Ed, Fred*}$\rangle$, $\langle$*AC, 3,* {*Bob, Ed, Fred*}$\rangle$, $\langle$*AE, 3,* {*Bob, Carol, Fred*}$\rangle$, $\langle$*BC, 3,* {*Don, Ed, Fred*}$\rangle$, $\langle$*BE, 3,* {*Alice, Don, Fred*}$\rangle$, *and* $\langle$*CE, 3,* {*Bob, Don, Fred*}$\rangle$. *In other words, after applying this second set of map$_2$ and reduce$_2$ functions, BigFoP framework discovered six interesting "following" patterns—in the form of pairs of frequently followed social entities—namely,* {*A,B*}, {*A,C*}, {*A,E*}, {*B,C*}, {*B,E*} *and* {*C, E*}, *that are followed by 2, 3, 3, 3, 3 and 3 followers, respectively. Of course, each of these six followee pairs is followed by at least 2 followers (the user-specified minimum frequency threshold).* ■

6.4.3 Subsequent Sets of Map-Reduce Functions in BigFoP

So far, BigFoP has found interesting "following" patterns in the form of (i) *individual* frequently followed social entities as well as (ii) *pairs* of frequently followed

social entities. BigFoP then applies *similar* sets of map and reduce functions to find triplets, quadruplets, quintuplets and higher (i.e., k-tuplets for $k \geq 3$) of frequently followed social entities:

$$\begin{aligned} \text{map}_{k\geq 3}\text{:}\quad & \langle \text{interesting followee } (k-1)\text{-tuplet } P, \text{ its follower information} \rangle \\ & \mapsto \langle \text{follower, followee } k\text{-tuplet} \rangle, \end{aligned} \tag{6.5}$$

with a detailed definition as follows:

for each $P \in \langle P, \_, \text{list}[P] \rangle$ emitted by reduce_{k-1} **do**
  for each follower $\in \text{list}[P]$ **do**
    for each $\langle \text{follower, followee} \rangle \in E$ of social network $G{=}(V, E)$ **do**
      if isRelevant(followee, P)
      then emit $\langle \text{follower}, P \cup \{\text{followee}\}, 1 \rangle$.

Again, isRelevant(followee, P) is a Boolean function checking the relevance (e.g., consistence to the mining order) of followee with respect to P. Since reduce_2 can be considered as an instance of the $\text{reduce}_{k\geq 2}$ function, the latter can be defined in a way very similar to that for reduce_2 as shown below:

$$\begin{aligned} \text{reduce}_{k\geq 2}\text{:}\quad & \langle \text{followee group, list of common followers} \rangle \\ & \mapsto \text{list of } \langle \textit{interesting} \text{ followee group, follower information} \rangle, \end{aligned} \tag{6.6}$$

with a detailed definition as follows:

for each $P \in \langle \_, \text{followee group } P, \_ \rangle$ emitted by map_{k-1} **do**

set counter[P] = 0;

set list[P] = {};

for each follower $\in \langle$follower, P, 1$\rangle$ emitted by map$_{k-1}$ **do**

counter[P] = counter[P] + 1;

list[P] = list[P] $\cup$ {follower};

if counter[P] $\geq$ user-specified min frequency threshold

then emit $\langle P$, counter[P], list[P]$\rangle$.

This results in (i) a list of followers and (ii) its count for each *interesting/popular* followee group P. See Example 6.5.

Example 6.5 *Continue with Example 6.4. For popular followee group AB (followed by two followers Ed & Fred), the map$_3$ function emits three* relevant *followees: {⟨Ed, ABC, 1⟩, ⟨Fred, ABC, 1⟩, ⟨Fred, ABE, 1⟩}. Then, for popular followee group AC (followed by three followers Bob, Ed & Fred), the map$_3$ function emits two* relevant *followees: {⟨Bob, ACE, 1⟩, ⟨Fred, ACE, 1⟩}. Similarly, for popular followee group BC (followed by three followers Don, Ed & Fred), the map$_3$ function emits two* relevant *followees: {⟨Don, BCE, 1⟩, ⟨Fred, BCE, 1⟩}.*

Afterwards, by applying the reduce$_3$ function, BigFoP discovers the following three interesting "following" patterns {A, B, C}, {A, C, E} and {B, C, E} with their associated lists and number of followees as ⟨ABC, 2, {Ed, Fred}⟩, ⟨ACE, 2, {Bob, Fred}⟩, and ⟨BCE, 2, {Don, Fred}⟩.

Based on the results returned by the reduce$_3$ function, BigFoP applies map$_4$ but returns nothing because there is no relevant quadruplet of frequently followed social entities. This completes the mining process for interesting "following" patterns from

our illustrative example social network. Note that key concepts and steps illustrated in this example are applicable to any big social network. ■

6.5 The EFoP-miner Framework

In my EFoP-miner framework, I apply EB-mine to discover frequent "following" patterns from social networks. Recall from Chapter 5, the data structure of EB-mine contains three main structures: Compressed B-table, HI-Counter and VI-List structures. Here, in the EFOP-miner Framework, the Compressed B-table become the Compressed SocialTable, the HI-Counter structure stays the same, and the VI-List structure become UserList. These structures are explained as follows:

1. a **Compressed SocialTable** T, which is the main bitmap based structure, in which each row-represented by a compressed bit vector-contains information about one particular user (a follower and its followees) in a social network;

2. an **HI-Counter** (Horizontal Index Counter) vector of level k (where k is the cardinality of the discovered "following" patterns), which is an index counter list stored as a vector; and

3. a **UserList** (or VI-List) structure of level k. For each relevant followee u in the current level k, the bit u is set to 1. Otherwise, it is set to 0. As we have observed earlier the adapted WAH-compressed bitmaps allow for efficient traversal of all bits set to 1. For all intents and purposes, this data structure behaves like a list, but with the memory footprint of a compressed bit vector.

This enables us to efficiently extract relevant users in a large social network without the need to check if they belong in the current projected data-base.

Let us consider an example social network with 930 social networking pages that can be followed, and we are analyzing the following information for 6 users (as a small example for the purpose of illustration).

To discover "following" patterns, we scan the interesting portion of a social network and create our bitwise table, a set of bit vectors for each social entity representing which of many followees they follow. We can also fill in the corresponding column sum. See Figure 6.2:

	followers	followees										
		0	...	15	...	40	...	50	...	92	...	646
Uncompressed SocialTable	Alice	0		1		0		0		1		0
	Bob	1		0		1		0		1		0
	Carol	1		0		0		0		1		0
	Don	0		1		1		0		1		0
	Ed	1		1		1		1		0		0
	Fred	1		1		1		0		1		0
Column sum		4		4		4		1		5		0

Figure 6.2: The Original SocialTable

First, we scan through the dataset and create the bitwise table, a set of bit vectors for each of the social entities representing which of 930 social networking pages they follow. We then compress each bit vector in the table. When adapting the WAH-compression scheme, we split bit vectors into 31-bit words to store the information. See Figure 6.3:

In cases where vectors are not 31-bits, we pad them with 0s.

	Index	**0**	**1**	**2**
Alice:	***FillMap***	0	1	0
	Array	0008000	80000001	40000000
Bob:	***FillMap***	0	0	0
	Array	0000001	00000200	40000000
Carol:	***FillMap***	0	1	0
	Array	0008001	80000001	40000000
Don:	***FillMap***	0	0	0
	Array	0008000	00000200	40000000
Ed:	***FillMap***	0	0	
	Array	0008001	00080200	
Fred:	***FillMap***	0	0	0
	Array	0008001	00000200	40000000

Figure 6.3: The Compressed SocialTable

Note that we only need to store a total of seventeen 32-bit words (i.e., 544 bits) instead of 6×930 bits $= 5580$ bits for all six social entities with a total of 930 social networking pages in the domain. In other words, our compressed bit vectors consume far less memory compared with the uncompressed counterparts. Moreover, the compressed bit vector eliminates problems associated with data sparsity. Recall that, by using CLZ and CTZ instructions, we can compute the bit position for easy random access.

Then, for each row r of this compressed SocialTable, we create a top-level userList. Specifically, we record the "row cutting index" (i.e., the column index for the first occurrence of a "1" bit in row r) in the userList. In Figure 6.4 the column sum is a vector that stores the count of "1"s in each column of the SocialTable. For example, the number of "1"s in column 2 (for social networking page 15) of column sum is 4, which means that four social entities are following page 15. Notice that followees 50

and 646 had frequency support values of 1 and 0 (i.e., less than user specified minsup of 2) in the level-0 SocialTable, they are not further considered (i.e., all entries in binary columns 50 and 646 can be ignored).

Col. 0	$\mathbf{6C}_{(\mathbf{16})} = 0110\ 1100_{(2)}$	Rows 2, 3, 5, 6 (Bob, Carol, Ed, Fred)
Col. 15	$\mathbf{90}_{(\mathbf{16})} = 1001\ 0000_{(2)}$	Rows 1, 4 (Alice, Don)

Figure 6.4: UserList for Level 0

The UserList, on the other hand, contains several rows of social entities first appearance. For example, in Figure 6.4, the UserList captures the information: (i) rows 2, 3, 5 and 6 (representing four followers Bob, Carol, Ed and Fred) contain lists of social entities that start with page 15; (ii) rows 1 and 4 (representing two followers Alice and Don) contain lists of social entities that start with page 15.

The social network mining process starts from the first row of the top-level UserList as shown in Figure 6.4. By using the "column cutting index" and "row cutting indices", we can "cut" the SocialTable into a smaller piece (as shown in Figure 6.5), to which the mining process for discovering frequent "followed" groups with prefix "page 0" can be applied. As shown in Figure 6.5, the original SocialTable is cut into the smaller section in the right hand side, named level-1 SocialTable.

Then, the next step is to construct a level 1-column sum and level 1-VI-List structure from the level-1 SocialTable. As shown in Fig. 5, the newly constructed level-1 HI-Counter structure records all the column counts of the level-1 SocialTable. These counts are frequency support values of all followed patterns with prefix "page 0". In this example, as we proceed to this step, we have frequency support values for "followed" patterns: "{social networking pages 0, 15}: 2" (which represent social

	Pages 0	15	40	50	92	646
Alice	0	1	0	0	1	0
Bob	1	0	1	0	1	0
Carol	1	0	0	0	1	0
Don	0	1	1	0	1	0
Ed	1	1	1	1	0	0
Fred	1	1	1	0	1	0

Figure 6.5: A level-1 SocialTable after cutting followee 0

networking pages {0, 15} followed by two social entities), "{pages 0, 40}: 3, and {pages 0, 92}: 3".

After constructing the level-1 HI-Counter structure, we then construct the level-1 VI-List. See Figure 6.6, which shows the level-1 VI-List structure that contains three rows.

Column 15: {Page 0, 15}	$\mathbf{C_{(16)}} = 0000\ 1100_{(2)}$	Rows 5 6 (Ed, Fred)
Column 40: {Page 0, 40}	$\mathbf{40_{(16)}} = 0100\ 0000_{(2)}$	Rows 2 (Bob)
Column 92: {Page 0, 92}	$\mathbf{20_{(16)}} = 0010\ 0000_{(2)}$	Rows 3 (Carol)

Figure 6.6: The level-1 VI-List structure for page 0

Based on this level-1 VI-List structure, we learn that we can construct a level-2 SocialTable (for {pages 0, 15}) and its associated level-2 HI-Counter structure, and level 1+2-VI-List structure. Then, our algorithm returns the result ("{pages 0, 15, 40}: 2"), which represents that group {pages 0, 15, 40} are followed by two social entities. At this point, we observe that no more frequent result can be mined from the remainder of the level-2 SocialTable for {pages 0, 15}. Therefore, the recursive process backtracks to the previous level. Afterwards, the algorithm checks if the

previous VI-List structure has more rows. If no, the algorithm returns. If yes, the algorithm checks if all other rows in the previous VI-List structure have the same "cutting column index" as the current VI-List rows have. If yes, update the new row in previous VI-List structure. Consider the example to illustrate this process in detail. After processing the first row of level-1 VI-List structure, there are two more rows remaining in the level-1 VI-List structure. Notice that, the cutting column index in the level-1 VI-List[1] is the same as in the level-2 VI-List[0], therefore we need to update the level-1 VI-List[1]. After that, the algorithm continues with the new level 1-VI-List structure. The result of this step is {pages 0, 40, 92} are followed together by only one social entity). As such a group is only followed by one social entity, which is less than the user-specified minsup. Hence, such a group is in-frequent (or not too popular). The algorithm follows the above process. Consequently, we obtain our mining result.

6.6 Summary

In this chapter, I introduced four frameworks FoP-miner, ParFoP-miner, BigFoP, and EFoP-miner, which applied B-mine, PB-mine, MRB-mine, and EB-mine to discover frequent "following" patterns in social networks (where frequent "following" patterns are defined in Section 2.3). As one of the most important parts of this thesis, these frameworks applies my algorithms to real-life applications. The evaluation of these frameworks will be presented in Chapter 7.

Chapter 7

Evaluation

In this chapter, I first present the evaluation results of the B-mine and EB-mine algorithms together in Section 7.1. Moreover, I compare my algorithms (both B-mine and EB-mine) with some existing frequent pattern mining algorithms in terms of both runtime and memory usage. Then, in Section 7.2, I present evaluations for both PB-mine (Parallel B-mine) and MRB-mine (MapReduce-based B-mine) as the ParFoP-miner framework and BigFoP framework for social network mining.

7.1 B-mine and EB-mine Evaluation

7.1.1 Analytical Evaluation

In terms of analytical evaluation, I analyze the memory space consumption of B-mine when compared with related works (i.e., FP-growth algorithm [HPY00]). Recall from Chapter 2 that FP-growth is a tree-based divide-and-conquer approach to mine frequent patterns from shopper market basket datasets. The algorithm first builds a

top-level FP-tree to capture important contents of the dataset in the FP-tree. The number of tree nodes is theoretically bounded above by the number of occurrences of all items (say, $O(N_{occurrence})$) in the dataset. Practically, due to tree path sharing, the number of tree nodes (say, $O(N_{tree}) < O(N_{occurrence})$) is usually smaller than the upper bound. However, during the mining process, multiple smaller sub-trees need to be constructed. Specifically, for a top-level FP-tree with depth d, it is not unusual for $O(d)$ sub-trees to coexist with the top-level tree, for a total of $O(d \times N_{tree})$ nodes. In contrast, on the surface, my B-table may appear to take up more space (due to the lack of tree path sharing). The B-table contains $O(N_{occurrence})$ entries. However, it is important to note that each entry in my B-table is just *a single bit*, instead of an integer for an item ID. In other words, the B-table requires $\frac{O(N_{occurrence})}{8}$ bytes of space. Moreover, unlike FP-growth, we do not need to build any additional copies of the B-table because the same B-table is used throughout the entire mining process. Thus, our B-table requires $\frac{O(N_{occurrence})}{8} \ll O(d \times N_{tree})$ bytes in FP-growth.

Recall from Chapter 3, the data structures used by B-mine consists of three parts: (i) a B-table, (ii) an HI-counter, and (iii) a VI-list. In the above analysis, I focused on B-table. How about VI-lists and HI-counters? For any frequent patterns of length k (i.e., groups of k frequently appearing items), we need to create $O(k)$ VI-lists and HI-counters. Note that FP-growth also creates $O(k)$ header tables for the $O(k)$ sub-trees. In other words, the amount of space required by VI-lists and HI-counters is the same to that by header tables. However, the size of each VI-list structure and HI-counter structure, together, are much less than sub-tree that constructed by FP-Growth algorithm at any recursion level. Moreover, the B-table requires much

less space than all FP-trees. Overall, the data structures of B-mine consume less memory space than FP-trees & their associated header tables (used in the FP-growth algorithm). Our data structure of B-mine is a *space-efficient data structure*. For instance, when using the SNAP ego-Facebook dataset (described in the next section), the amount of memory space consumed by the data structures of B-mine is about 29% of that consumed by FP-trees & their associated header tables used in the FP-growth algorithm. Similarly, the amount of memory space consumed by the data structures of B-mine used in B-mine is about 16% of that consumed by FP-trees & their associated header tables used in the FP-growth algorithm.

Recall from Chapter 5 that, compare with B-mine, EB-mine enhanced the algorithm by applying the compression on the B-table. Analytically, EB-mine finds the index of the previous or next "1" bit in O(1) (cf. O(n) where n is the size of bit vector in the uncompressed binary vectors using in B-mine). Similarly, EB-mine finds the value of all bits (i.e., traversal) in O(1) due to random access functions (cf. O(n) in B-mine). Moreover, our solution applies bitwise AND, OR, NOT operations on only j adapted WAH-compressed words, whereas B-mine applies more often (e.g., $n/32 \gg j$ words). Which, leads to the conclusion that EB-mine perform even better than B-mine in both runtime and memory consumption. In the next several sections, I do an empirical evaluation to see if this analytical evaluation is correct.

7.1.2 Empirical Evaluation

In terms of empirical evaluation, I compare the performance of our B-mine and EB-mine algorithms with four existing classic frequent pattern mining algorithms:

FP-growth [HPY00], H-mine [PHL+01], VIPER [SHS+00], and Eclat [Zak00]. For datasets, I used (i) synthetic datasets, which are generally sparse and are generated by the IBM Quest synthetic data generator [AS94], and (ii) Retail real-life dataset [BSVW99] from the Frequent Itemset Mining Dataset Repository (`http://fimi.ua.ac.be/data/`).

All experiments were run in a time-sharing environment in a 1 GHz machine. I ran the algorithms 10 times, and excluded the runs with maximum and minimum runtimes. Hence, the reported figures are based on the average of 8 runs. Runtime includes CPU and I/O operations; it includes the time for both the construction of the data structure of B-mine and the mining of frequent patterns.

Figures 7.1 and 7.2 show the result on two sparse IBM synthetic datasets (one contains 10K transactions and the other contains 100K transactions) when we vary user-specified frequency threshold values. When the user-specified frequency threshold was low, H-mine and VIPER did not perform very well. Eclat showed its advantages when handling sparse datasets. My B-mine and EB-mine generally performed better than the other four algorithms.

Similarly, Figure 7.3 shows the results on the dense real-life retail dataset [BSVW99] from the Frequent Itemset Mining Dataset Repository. This dataset captures 88,163 transactions on the purchases of 16,470 unique items by 5,133 customers. My B-mine and EB-mine, again, gave the best performance over the other four existing algorithms.

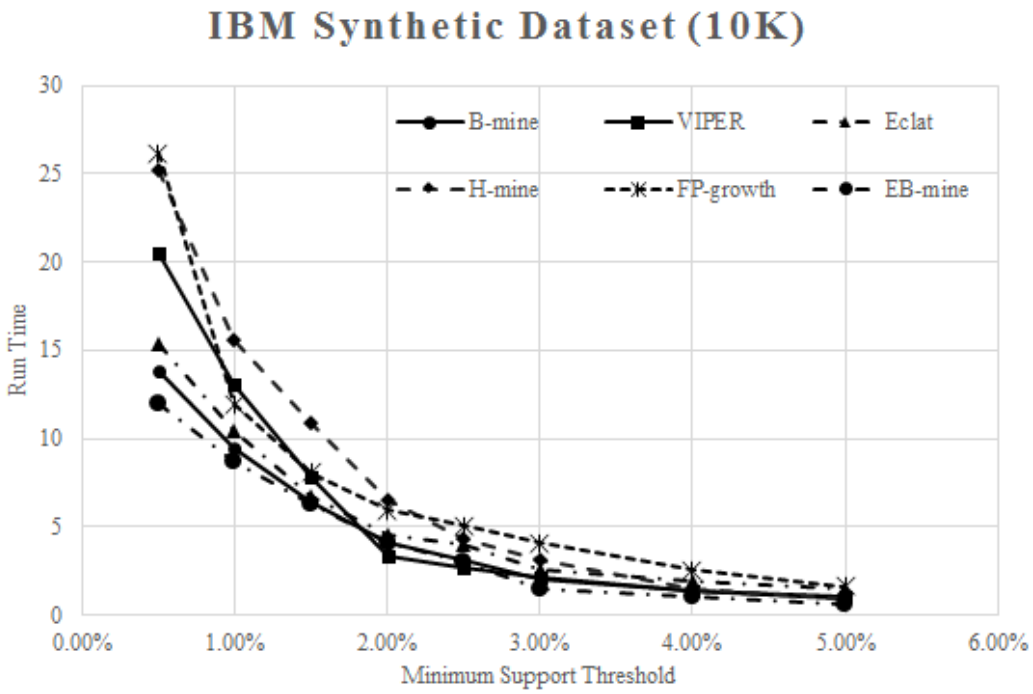

Figure 7.1: Experimental results on IBM synthetic datasets with 10K transactions.

7.2 PB-mine and MRB-mine Evaluation

7.2.1 PB-mine evaluation in the ParFoP-miner Framework

To handle big data, I introduced PB-mine, as a parallel version of B-mine in Section 4.1. A social network mining framework–ParFoP-miner– was introduced in Section 6.3, which, applies PB-mine to social network for finding "following" patterns.

Figure 7.4 shows the runtime improvement provided by our parallel framework ParFoP-miner (when using only two processors) over our serial algorithm FoP-miner (which applies B-mine to find "following" patterns).

As we can see from Figure 7.4, the parallel method improved the performance.

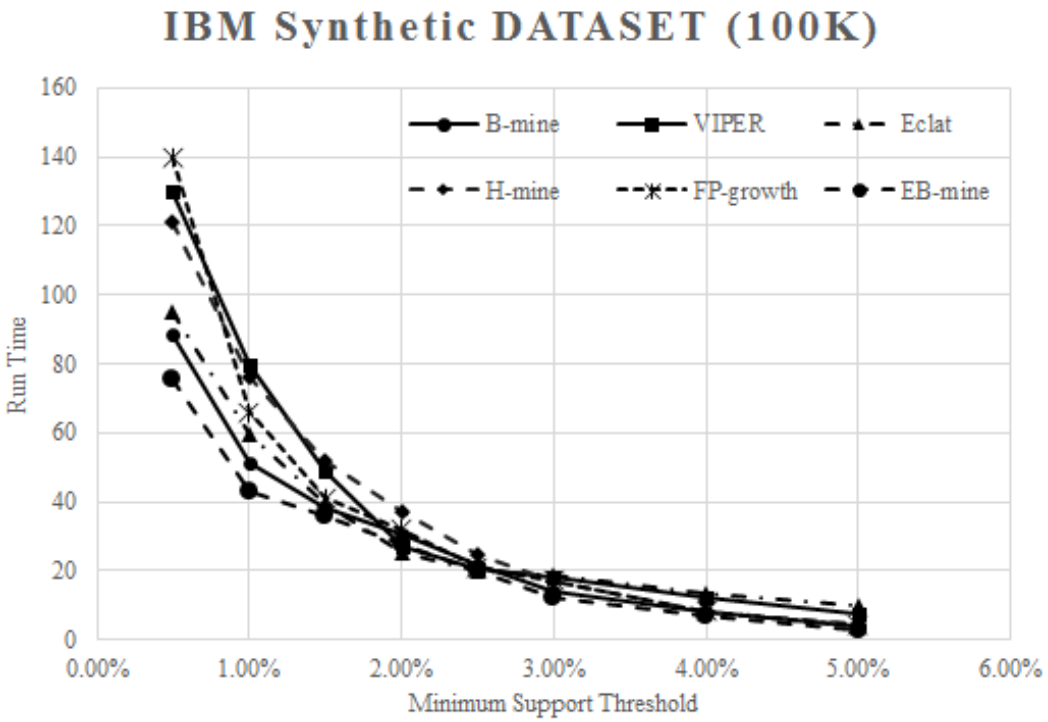

Figure 7.2: Experimental results on IBM synthetic datasets with 100K transactions.

The gap between the two algorithms would be bigger if we used more processors in parallel mining.

7.2.2 MRB-mine evaluation in the BigFoP Framework

In Section 6.4, I introduced another framework to discover "following" patterns from social networks database using the BigFoP Framework, in which, MRB-mine (Section 4.2) was applied.

To discover "following" patterns, BigFoP takes advantage of the MapReduce model. The input social data are divided into several partitions (subproblems) and

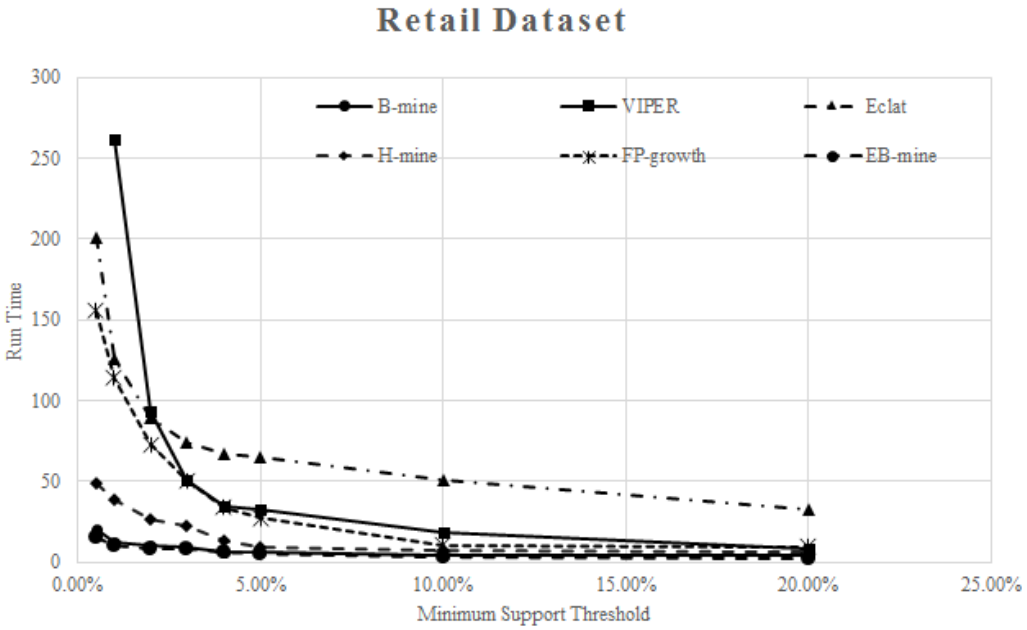

Figure 7.3: Experimental results on retail real-life datasets.

assigned to different processors. Each processor executes the map_k and reduce_k functions (for $k \geq 1$). On the surface, one might worry that lots of communications or exchanges of information are required among processors. Fortunately, due to the divide-and-conquer nature of our big social network data analytics solution of discovering "following" patterns, once the original big social network is partitioned and assigned to each processor (e.g., one processor is assigned the followers of A, another is assigned the followers of B, a third one is assigned the followers of C), each processor handles the assigned data without any reliance on the results from other processors. As observed from the above examples, the processor assigned for the followers of a popular followee can apply the subsequent sets of map and reduce functions on data

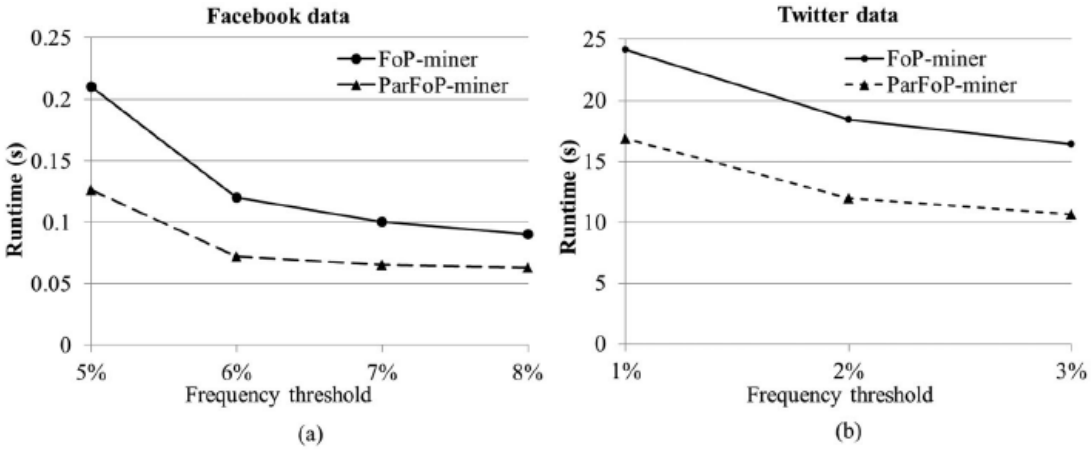

Figure 7.4: Experimental results of ParFoP-miner on social network datasets.

emitted by that processor. For example, a processor applies map_1 and reduce_1 to find popular followee A. That processor can then apply map_2 on the data emitted by reduce_1 from that processor to find popular followee group AB (i.e., group containing A). Similarly, the processor applies map_3 on the data emitted by reduce_2 from the same processor to find subsequent popular followee group ABC. Without the need of extra communications and exchanges of data among processors, the BigFoP discovers all interesting "following" patterns more efficiently compare with the non-parallel original version. Moreover, if a partition of the big social network is too big to be handled by a single processor, the BigFoP further sub-divides that partition so that the resulting sub-partitions can be handled by each of multiple processors.

Furthermore, due to the divide-and-conquer nature of our big social network data analytics solution of discovering "following" patterns, the amount of data input for the map_k and reduce_k functions monotonically decreases as the size of the popular group of k followees increases. The BigFoP framework discovers all interesting "following"

patterns in a space effective manner.

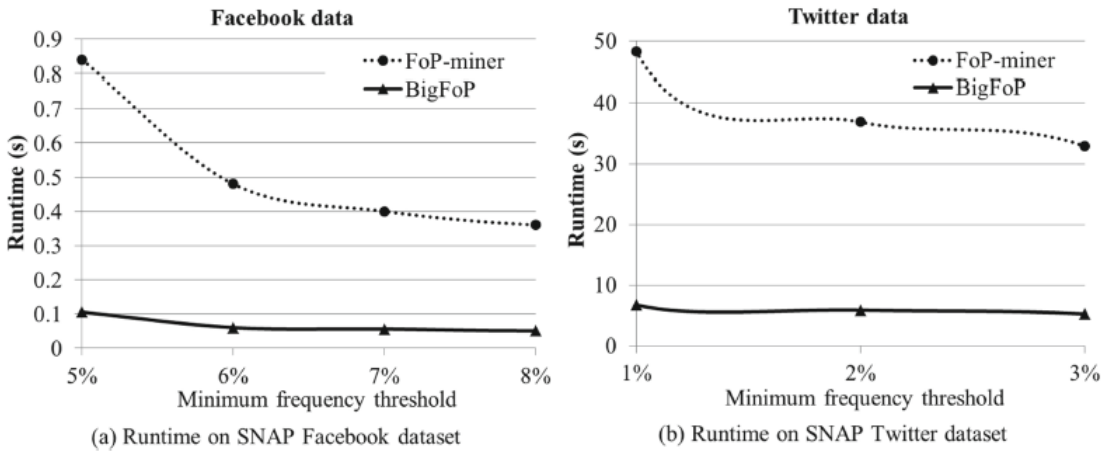

Figure 7.5: Experimental results of BigFoP on social network datasets.

All experiments were run using either (i) a single machine with an Intel Core i7 4-core processor (1.73 GHz) and 8 GB of main memory running a 64-bit Windows 7 operating system, or (ii) the Amazon Elastic Compute Cloud (EC2) cluster—specifically, 11 High-Memory Extra Large (m2.xlarge) computing nodes. I implemented both the existing FoP-miner algorithm and my BigFoP framework in the Java programming language. The stock version of Apache Hadoop 0.20.0 was used. The results shown in Figure 7.5, in which the x-axis shows the user-specified minimum frequency threshold (in percentage of the number of social entities) expressing the interestingness of the mined patterns, are based on the average of multiple runs. Runtime includes CPU and I/Os in the mining process of interesting "following" patterns. In particular, Figure 7.5(a) shows that BigFoP provided a speedup of about 8 times when compared with FoP-miner when mining the SNAP Facebook dataset. Higher speedup is expected when using more processors. Figure 7.5(b) shows a similar

result for the SNAP Twitter dataset.

7.2.3 EB-mine evaluation in the EFoP-miner Framework

Recall from Section 6.5, EFoP-miner Framework applies EB-mine for discovering frequent "following" patterns in social networks. In this section I present the evaluation of this framework. As in the previous two sections, I use FoP-miner (which applies the original B-mine algorithm) as a base line for assessing compression.

Recall that analytically, EFoP-miner can find the index of the previous/next 1 bit in O(1) time (cf. O(n) where n is the size of the uncompressed bit vectors used in FoP-Miner). Similarly, our solution finds the value of all bits (i.e., traversal) in O(1) time due to random access functions (See Section 5.1.1) (cf. O(n) in FoP-Miner). Moreover, my solution applies bitwise AND, OR, NOT operations on only j adapted WAH-compressed words, whereas FoP-Miner applies more often (e.g., $n/32 \gg j$ words).

Empirically, I compared the performance of the EFoP-miner framework with FoP-Miner by using the Twitter dataset from the Stanford Large Network Dataset Collection (http://snap.stanford.edu/data/).

For this evaluation, again, all experimental results are based on the average of 10 runs. Runtime includes CPU and I/O (refer to Figure 7.6). The storage savings varies depending on the sparsity of data–we naturally experience larger savings on very sparse datasets with long runs of 0s, or on very dense datasets with large runs of 1s but in the absolute worst case, we will store the same bit vector. We will usually be able to compress at least some part of the bit vector. The WAH-compressed

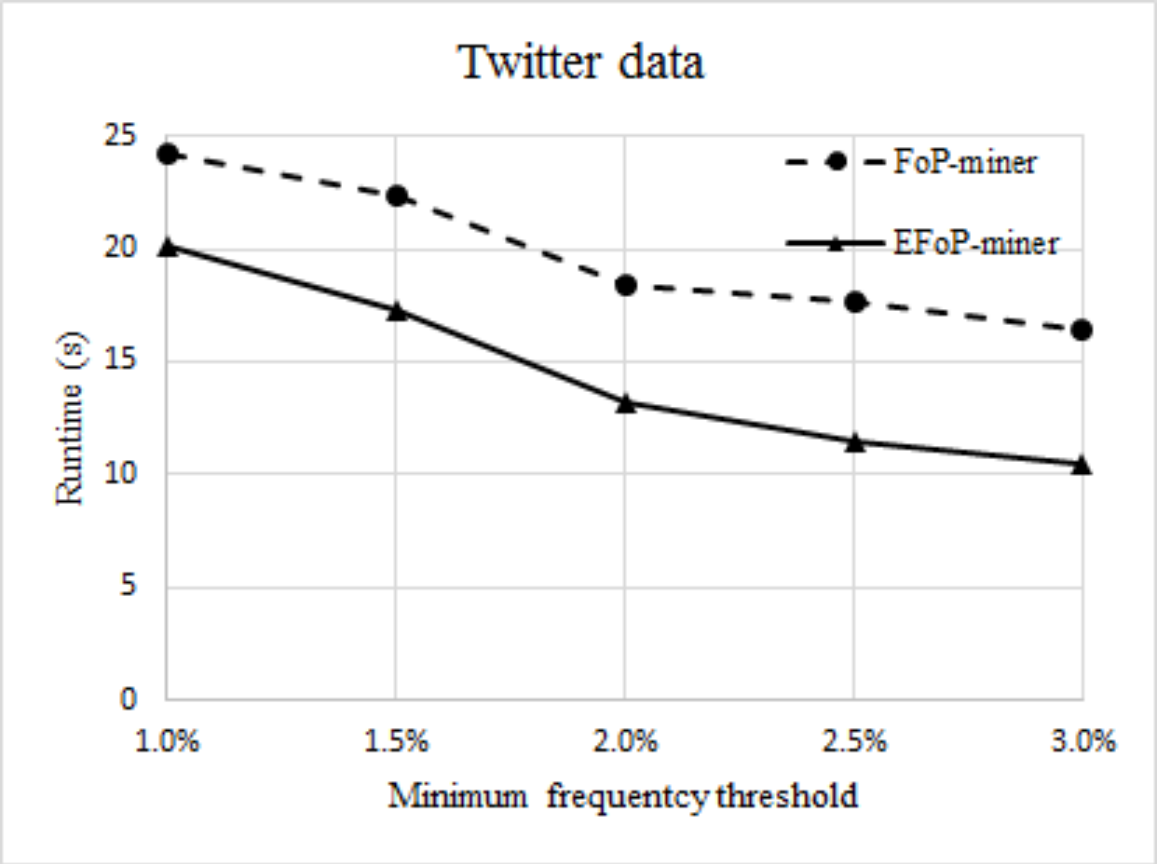

Figure 7.6: Experimental results of EFoP-miner on social network dataset.

transactions of our solution also allowed for fast extraction of relevant bits (i.e., those equal to 1), which dramatically improved the process of support counting. Overall, my solution performs faster on sparse datasets such as social network data and consumes significantly less memory.

7.3 Summary

In this chapter, I evaluated all of my proposed algorithms and frameworks. Both B-mine and EB-mine were evaluated by comparing them with existing algorithms. PB-mine and MRB-mine were evaluated based on a real-life problem, namely social network mining using the ParFoP-miner Framework and BigFoP Framework.

The evaluation of B-mine and EB-mine was done together both analytically (Section 7.1.1) and empirically (Section 7.1.2). For analytical evaluation, I analyzed the memory usage of both algorithms against existing work. As the construction of the B-table in B-mine was well designed, the required memory use of B-mine was shown to be less than existing work (e.g., Apriori and FP-growth). Furthermore, by applying compression, EB-mine further decreased the memory usage to be a more space-efficient algorithm. For empirical evaluation, I compared the runtime of both B-mine and EB-mine against four existing algorithms (namely, FP-growth, H-mine, VIPER and Eclat). The evaluation result showed advantages of my algorithms on runtime on different datasets.

For PB-mine and MRB-mine, as they were used in ParFoP-miner Framework and BigFoP Framework for real-life problem (finding frequent "following" patterns in social network mining). They are both evaluated in a more piratical and application approach. By comparing both the runtime and scalability against FoP-miner Framework (which applies B-mine to find frequent "following" patterns), both the ParFoP-miner Framework and the BigFoP Framework showed improvements in terms of runtime.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

As one of the most important research topics in data mining, frequent pattern mining aims to discover implicit, previously unknown, and potentially useful knowledge from data. In the past decades, algorithms such as Apriori, FP-growth, and H-mine have been created. While these algorithms are showing their advantages and benefit data mining research, they also suffer from certain drawbacks/disadvantages.

In this Ph.D. thesis, I first proposed an algorithm, called B-mine, which is able to overcome several existing drawbacks and achieve better performance when compared with existing algorithms. After that, I improved B-mine to be able to handle big data mining. This improvement was done in two different research directions:

1. Parallel and MapReduce, which focused on using parallel environments to handle big volumes of data. Two algorithms were designed as outcomes: PB-mine (Section 4.1) and MRB-mine (Section 4.2)

2. Compression and Enhancement, which focused on compressing the data structure and thereby improving the space efficiency of the mining process. The algorithm EB-mine (Chapter 5) was designed to explore this research direction.

Furthermore, as another important part of my Ph.D. thesis, I applied my proposed algorithms to real-life application—social network mining—to discover frequent "following" patterns. Four frameworks were designed: (1) FoP-miner, (2) ParFoP-miner, (3) BigFoP, and (4) EFoP-miner using each of the four created algorithms.

For evaluation, I conducted both analytical and empirical evaluations. Compared with existing algorithms (e.g., FP-growth, H-mine, etc.) my B-mine algorithm, as a fundamental algorithm of this thesis, is able to achieve better performance. Furthermore, I compared with my original B-mine algorithm, the evaluation results for PB-mine, MRB-mine and EB-mine to show their improvements considering different research components (i.e., Parallel computing, MapReduce, WAH data compression).

Recall that in Section 1.2, I asked several questions about this research. In this thesis, I provided answers for all of them:

1. *Can we overcome these existing drawbacks (e.g., high I/O cost, sub-tree structure construction, constant hyperlinks updates) and achieve better performance?*

   Yes, my proposed B-mine algorithm and a family of algorithms extended from B-mine presented in this Ph.D. thesis can overcome these drawbacks, and the evaluation showed that my B-mine algorithm can achieve a better performance.

2. *How to design a new data structure and a corresponding algorithm that will help us improve the performance?*

The B-mine algorithm consists of three main data structures: (1) B-table, (2) HI-Counter, and (3) VI-List. Details of these data structures were introduced in Chapter 3.

3. *Will the new algorithm applicable for big data mining?*

   Yes, B-mine algorithm presented in Chapter 3 is very suitable for big data mining for two reasons: (1) B-table is suitable for both parallel computing and MapReduce because of the data independency, and (2) the bitmap structure B-table is very suitable for data compression.

4. *What modifications to the new algorithm are needed for processing large volumes of data?*

   Some adjustments on the data structure of B-mine algorithm (e.g., B-table, VI-lists, and HI-counters) are required when we apply parallel computing, MapReduce and data compression. More details were introduced in Chapter 4 and Chapter 5.

5. *Can parallel/MapReduce be applied to the new algorithm?*

   Yes, parallel/MapReduce can be applied to my B-mine algorithm. The resulting algorithms PB-mine and MRB-mine was introduced in Chapter 4. Yes, they can. Two corresponding algorithms were designed, they are: PB-mine, and MRB-mine (Chapter 4).

6. *Can data compression be applied to the new algorithm?*

   Yes, data compression can be applied to my B-mine algorithm. The resulting algorithm EB-mine (which applies WAH compression) was introduced in

Chapter 5.

7. *How well would the above two methods improve the performance?*

   The Parallel and MapReduce methods presented in Chapter 4 improved the space-efficiency by distributing data to different processors, and improved the time-efficiency by distributing mining tasks to multiple processors. On the other hand, compression method improved the space-efficiency by compressing the bitmap into compressed bit vectors, and improved the time-efficiency by applying bit access in WAH compression.

8. *Will the algorithm still be able to achieve better performance when compare with existing work for these extended research areas?*

   Yes, the evaluation presented in Chapter 7 showed the improvements of these algorithms when compare with the original B-mine.

9. *Does the new algorithm applicable to real-life problems (e.g., social network mining)?*

   Yes, in Chapter 6, I introduced the social network mining real-life application of my proposed algorithms.

10. *What problem can it be applied to solve?*

    My algorithms presented in this Ph.D. thesis can be applied to many different real-life problems. For instance, in this thesis, I introduced one of them—discovering frequent "following" patterns in social network.

11. *How good can we solved this problem?*

    In this thesis, I proposed four frameworks: FoP-miner, ParFoP-miner, BigFoP,

and EFoP-miner in Chapter 6. The evaluation results showed the efficiency of these frameworks.

In conclusion, I successfully designed and developed a new frequent pattern mining algorithm, named B-mine, which can overcome some drawbacks/disadvantages that some existing frequent pattern mining algorithms are suffering from. Furthermore, I successfully further extended and enhanced the original B-mine algorithm for the problem of big data mining. The resulting algorithms are: (1) PB-mine, which applies parallel computing, (2) MRB-mine, which uses MapReduce, and (3) EB-mine, which compresses the data structure and enhances the performance. Furthermore, I applied my new algorithms to the real-life application of social network mining to discover frequent "following" patterns among social network users, for which, four frameworks were implemented, they are: (1) FoP-miner framework, which applies the original B-mine, (2) ParFoP-miner framework, which applies the PB-mine algorithm, (3) BigFoP framework, which applies the MRB-mine algorithm, and (4) EFoP-miner framework, which applies the EB-mine algorithm. Finally, the evaluation results showed the efficiency and practicality of my algorithms and frameworks.

8.2 Future Work

For the future work arising from this Ph.D. thesis, I plan to further extend this research in at least the following two directions: (1) Uncertain data mining, and (2) Data stream mining.

8.2.1 Uncertain Data Mining

In this thesis, I focused on mining frequent patterns from *precise datasets*, in which users definitely know whether an item is present in, or absent from, a transaction in the data. However, there are situations in which users are uncertain about the presence or absence of items. For example, due to dynamic errors (e.g., inherited measurement inaccuracies, sampling frequency), streaming data collected by sensors may be uncertain. As such, users may highly suspect but cannot guarantee that an item x is present in a transaction t_i. The uncertainty of such suspicion can be expressed in terms of existential probability $P(x, t_i) \in (0, 1]$, which indicates the likelihood of x being present in t_i in probabilistic data. With this notion, every item in t_i in (static databases or dynamic streams of) precise data can be viewed as an item with a 100% likelihood of being present in t_i. A challenge of handling these uncertain data is the huge number of "possible worlds" (e.g., there are two "possible worlds" for an item x in t_i: (i) $x \in t_i$ and (ii) $x \notin t_i$). Given q independent items in all transactions, there are $O(2^q)$ "possible worlds" [Leu11]. To discover frequent patterns from such type of data with uncertainty is called *uncertain data mining*.

Currently, the B-mine algorithm (and all its extension algorithms) can only handle *precise datasets*. I am planning to extend this algorithm for uncertain data mining. To do so, there are many challenges. One of the biggest challenges is to represent uncertainty using bitmap structure. As we know bitmap is good for marking items (or absents of items) in transactions. However, it is very difficult to represent the extra information (i.e., uncertainty) in uncertain dataset using bitmap.

8.2.2 Data Stream Mining

The algorithms I have discussed so far mine from traditional *static databases*. Nowadays, the automation of measurements and data collection is producing tremendously huge volumes of data. For instance, the development and increasing use of a large number of sensors (e.g., electromagnetic, mechanical, and thermal sensors) for various real-life applications (e.g., environment surveillance, manufacturing systems) have led to data streams [MSL08, Cuz09, CGWD10]. To discover frequent patterns from such dynamic data streams is called *data stream mining*.

In general, mining frequent patterns from dynamic data streams [JG06, GBK10] is more challenging than mining from traditional static transaction databases due to the following characteristics of data streams:

1. *Data streams are continuous and unbounded.* As such, we no longer have the luxury to scan the streams multiple times. Once the streams flow through, we lose them. We need some techniques to capture important contents of the streams.

2. *Data in the streams are not necessarily uniformly distributed.* As such, a currently infrequent pattern may become frequent in the future, and vice versa. We have to be careful not to prune infrequent patterns too early; otherwise, we may not be able to get complete information such as supports of some patterns (as it is impossible to recall those pruned patterns).

I had research experiences on data stream mining since this is one of the most important topics in my M.Sc. thesis. Now, as the second future research direction,

I am planning to overcome these aforementioned challenges and extend my B-mine algorithm to handle dynamic data streams.

Bibliography

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD 1993), Washington, DC, USA*, pages 207–216. ACM, 1993.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994), Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann Publishers Inc., 1994.

[BSVW99] Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[CBS13] Alfredo Cuzzocrea, Ladjel Bellatreche, and Il-Yeol Song. Data warehousing and olap over big data: Current challenges and future research directions. In *Proceedings of the 16th International Workshop on Data Warehousing and OLAP (DOLAP 2013), San Francisco, California, USA*, pages 67–70. ACM, 2013.

[CGWD10] Malu Castellanos, Chetan Gupta, Song Wang, and Umeshwar Dayal. Leveraging web streams for contractual situational awareness in operational BI. In *Proceedings of the 2010 International Conference on Extending Database Technology/International Conference on Database Theory (EDBT/ICDT 2010) Workshops, Lausanne, Switzerland*, pages 7:1–8. ACM, 2010.

[CLM14] Alfredo Cuzzocrea, Carson Kai-Sang Leung, and Richard Kyle MacKinnon. Mining constrained frequent itemsets from distributed uncertain data. *Future Generation Computer Systems*, 37:117–126, 2014.

[CSU13] Alfredo Cuzzocrea, Domenico Saccà, and Jeffrey D. Ullman. Big data: A research agenda. In *Proceedings of the 17th International Database Engineering & Applications Symposium (IDEAS 2013), Barcelona, Spain*, pages 198–203. ACM, 2013.

[Cuz09] Alfredo Cuzzocrea. CAMS: OLAPing multidimensional data streams efficiently. In *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2009), Linz, Austria*, pages 48–62. Springer-Verlag, 2009.

[CZLW15] Yifan Chen, Xiang Zhao, Xuemin Lin, and Yang Wang. Towards frequent subgraph mining on single large uncertain graphs. In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 41–50, 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DLJ16] Edson Dela Cruz, Carson Kai-Sang Leung, and Fan Jiang. Mining 'following' patterns from big sparse social networks. In *Proceedings of the International Symposium on Foundations and Applications of Big Data Analytics (FAB 2016), San Francisco, CA, USA*, pages 923–930. ACM, 2016.

[EZ06a] Mohammad El-Hajj and Osmar R. Zaïane. Parallel bifold: Large-scale parallel pattern mining with constraints. *Distributed and Parallel Databases*, 20(3):225–243, 2006.

[EZ06b] Mohammad El-Hajj and Osmar R. Zaïane. Parallel leap: Large-scale maximal pattern mining in a distributed environment. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS 2006), Minneapolis, USA*, pages 135–142. IEEE, 2006.

[fac] Facebook newsroom – company info. `http://newsroom.fb.com/company-info/`. Accessed: 2016-09-30.

[FS16] Jaroslav M. Fowkes and Charles A. Sutton. A subsequence interleaving model for sequential pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA*, pages 835–844, 2016.

[GBK10] Anamika Gupta, Vasudha Bhatnagar, and Naveen Kumar. Mining closed itemsets in data stream using formal concept analysis. In *Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2010), Bilbao, Spain*, pages 285–296. Springer-Verlag, 2010.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Records*, 29(2):1–12, May 2000.

[HSB+15] Kun He, Yiwei Sun, David Bindel, John E. Hopcroft, and Yixuan Li. Detecting overlapping communities from local spectral subspaces. In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 769–774, 2015.

[IJYZ12] Ayad Ibrahim, Hai Jin, Ali A. Yassin, and Deqing Zou. Towards privacy preserving mining over distributed cloud databases. In *Proceedings of the 2nd International Conference on Cloud and Green Computing (CGC 2012), Xiangtan, Hunan, China*, pages 130–136. IEEE Computer Society, 2012.

[IZ12] Leila Ismail and Liren Zhang. Modeling and performance analysis to predict the behavior of a divisible load application in a cloud computing environment. *Algorithms*, 5(2):289–303, 2012.

[JG06] Nan Jiang and Le Gruenwald. Research issues in data stream association rule mining. *SIGMOD Records*, 35(1):14–19, March 2006.

[JKL08] Fan Jiang, Kyoji Kawagoe, and Carson K. Leung. Big social network mining for "following" patterns. In *Proceedings of the 8th International C* Conference on Computer Science & Software Engineering (C3S2E 2015), Yokohama, Japan*, pages 28–37. ACM, 2008.

[JL13] Fan Jiang and Carson Kai-Sang Leung. Stream mining of frequent patterns from delayed batches of uncertain data. In *Proceedings of the 15th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2013), Prague, Czech Republic*, pages 209–221. Springer-Verlag New York, Inc., 2013.

[JL14a] Fan Jiang and Carson Kai-Sang Leung. A business intelligence solution for frequent pattern mining on social networks. In *Proceedings of the 2014 IEEE International Conference on Data Mining Workshops (ICDM Workshops 2014), Shenzhen, China*, pages 789–796. IEEE, 2014.

[JL14b] Fan Jiang and Carson Kai-Sang Leung. Mining interesting "following" patterns from social networks. In *Proceedings of the 16th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2014), Munich, Germany*, pages 308–319. Springer, 2014.

[JLLP14] Fan Jiang, Carson Kai-Sang Leung, Dacheng Liu, and Aaron M. Peddle. Discovery of really popular friends from social networks. In *Proceedings of the 4th IEEE International Conference on Big Data and Cloud Computing (BDCloud 2014), Sydney, Australia*, pages 342–349, 2014.

[JLZ16] Fan Jiang, Carson Kai-Sang Leung, and Hao Zhang. B-mine: Frequent pattern mining and its application to knowledge discovery from social networks. In *Proceedings of the 18th Asia-Pacific Web Conference (APWeb 2016), Suzhou, China*, pages 316–328. Springer, 2016.

[Kej12] Arun Kejariwal. Big data challenges: A program optimization perspective. In *Proceedings of the 2nd International Conference on Cloud and Green Computing (CGC 2012), Xiangtan, China*, pages 702–707. IEEE Computer Society, 2012.

[KNR13] Arun Kumar, Feng Niu, and Christopher Ré. Hazy: Making it easier to build and maintain big-data analytics. *Queue*, 11(1):30:30–30:46, January 2013.

[KYWM15] Ying Kang, Bo Yu, Weiping Wang, and Dan Meng. Spectral clustering for large-scale social networks via a pre-coarsening sampling based nyström method. In *Proceedings of the 19th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD 2015), Ho Chi Minh City, Vietnam*, pages 106–118, 2015.

[LCJ13] Carson Kai-Sang Leung, Alfredo Cuzzocrea, and Fan Jiang. Discovering frequent patterns from uncertain data streams with time-fading and landmark models. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 8:174–196, 2013.

[Leu11] Carson Kai-Sang Leung. Mining uncertain data. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(4):316–329, 2011.

[LH13] Carson Kai-Sang Leung and Yaroslav Hayduk. *Database Systems for Advanced Applications: 18th International Conference, DASFAA 2013, Wuhan, China, April 22-25, 2013. Proceedings, Part I*, chapter Mining Frequent Patterns from Uncertain Data with MapReduce for Big Data Analytics, pages 440–455. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[LJ14] Carson Kai-Sang Leung and Fan Jiang. A data science solution for mining interesting patterns from uncertain big data. In *Proceedings of the 4th IEEE International Conference on Big Data and Cloud Computing (BDCloud 2014), Sydney, Australia*, pages 235–242. IEEE Computer Society, 2014.

[LJ15] Carson Kai-Sang Leung and Fan Jiang. Big data analytics of social networks for the discovery of "following" patterns. In *Proceedings of the 17th International Conference on Big Data Analytics and Knowledge*

Discovery (DaWaK 2015), Valencia, Spain, pages 123–135. Springer, 2015.

[LJPP16] Carson Kai-Sang Leung, Fan Jiang, Adam G. M. Pazdor, and Aaron M. Peddle. Parallel social network mining for interesting 'following' patterns. *Concurrency and Computation: Practice and Experience*, 28(15):3994–4012, 2016.

[LM14] Carson Kai-Sang Leung and Richard Kyle MacKinnon. Fast algorithms for frequent itemset mining from uncertain data. In *Proceedings of the 2014 IEEE International Conference on Data Mining (ICDM 2014), Shenzhen, China*, pages 893–898, 2014.

[LMJ14a] Carson Kai-Sang Leung, Richard Kyle MacKinnon, and Fan Jiang. Distributed uncertain data mining for frequent patterns satisfying anti-monotonic constraints. In *Proceedings of the 28th International Conference on Advanced Information Networking and Applications Workshops (WAINA 2014), Victoria, Canada*, pages 1–6. IEEE Computer Society, 2014.

[LMJ14b] Carson Kai-Sang Leung, Richard Kyle MacKinnon, and Fan Jiang. Reducing the search space for big data mining for interesting patterns from uncertain data. In *Proceedings of the 2014 IEEE International Congress on Big Data (BigData Congress 2014)*, pages 315–322. IEEE Computer Society, 2014.

[LML+16] Xiang Li, Milad Makkie, Binbin Lin, Mojtaba Sedigh Fazli, Ian Davidson, Jieping Ye, Tianming Liu, and Shannon Quinn. Scalable fast rank-1 dictionary learning for fmri big data analysis. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA*, pages 511–519, 2016.

[LT12] Carson Kai-Sang Leung and Syed Khairuzzaman Tanbeer. Mining popular patterns from transactional databases. In *Proceedings of the 14th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2012), Vienna, Austria*, pages 291–302, 2012.

[LTC14] Carson Kai-Sang Leung, Syed Khairuzzaman Tanbeer, and Juan J. Cameron. Interactive discovery of influential friends from social networks. *Social Netw. Analys. Mining*, 4(1):154, 2014.

[LXWC15] Liyuan Liu, Linli Xu, Zhen Wang, and Enhong Chen. Community detection based on structure and content: A content propagation perspective.

In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 271–280, 2015.

[Mad12a] Sam Madden. From databases to big data. *IEEE Internet Computing*, 16(3):4–6, May 2012.

[Mad12b] Sam Madden. From databases to big data. *IEEE Internet Computing*, 16(3):4–6, 2012.

[MSL08] George A. Mihaila, Ioana Stanoi, and Christian A. Lang. Anomaly-free incremental output in stream processing. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM 2008), Napa Valley, CA, USA*, pages 359–368. ACM, 2008.

[MWW+15] Robert Moskovitch, Colin Walsh, Fei Wang, George Hripcsak, and Nicholas P. Tatonetti. Outcomes prediction via time intervals related patterns. In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 919–924, 2015.

[NSK+16] Kazuya Nakagawa, Shinya Suzumura, Masayuki Karasuyama, Koji Tsuda, and Ichiro Takeuchi. Safe pattern pruning: An efficient approach for predictive pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA*, pages 1785–1794, 2016.

[PHL+01] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001), San Jose, CA, USA*, pages 441–448. IEEE, 2001.

[RG15] Emilee Rader and Rebecca Gray. Understanding user beliefs about algorithmic curation in the facebook news feed. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI 2015), Seoul, Republic of Korea*, pages 173–182. ACM, 2015.

[RZL15] Ashwin Rajadesingan, Reza Zafarani, and Huan Liu. Sarcasm detection on twitter: A behavioral modeling approach. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining (WSDM 2015), Shanghai, China*, pages 97–106. ACM, 2015.

[SAM15] Saber Salah, Reza Akbarinia, and Florent Masseglia. Fast parallel mining of maximally informative k-itemsets in big data. In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 359–368, 2015.

[SHS+00] Pradeep Shenoy, Jayant R. Haritsa, S. Sudarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. Turbo-charging vertical mining of large databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, (SIGMOD 2000), Dallas, Texas, USA*, pages 22–33. ACM, 2000.

[SW09] Michal Stabno and Robert Wrembel. RLH: bitmap compression technique based on run-length and huffman encoding. *Inf. Syst.*, 34(4-5):400–414, 2009.

[TLC14] Syed Khairuzzaman Tanbeer, Carson Kai-Sang Leung, and Juan J. Cameron. Interactive mining of strong friends from social networks and its applications in e-commerce. *Journal of Organizational Computing and Electronic Commerce*, 24(2-3):157–173, 2014.

[twi] Company about – twitter. `https://about.twitter.com/company`. Accessed: 2016-09-30.

[WD15] Dawei Wang and Wei Ding. A hierarchical pattern learning framework for forecasting extreme weather events. In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 1021–1026, 2015.

[wei] Weibo performance in q2 2016 – china internet watch. `https://www.chinainternetwatch.com/18681/weibo-q2-2016/`. Accessed: 2016-09-30.

[WKGL16] Hongjian Wang, Daniel Kifer, Corina Graif, and Zhenhui Li. Crime rate inference with big data. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA*, pages 635–644, 2016.

[WWX15] Longhui Wang, Yong Wang, and Yudong Xie. Implementation of a parallel algorithm based on a spark cloud computing platform. *Algorithms*, 8(3):407–414, 2015.

[XYX+15] Huang Xu, Zhiwen Yu, Hui Xiong, Bin Guo, and Hengshu Zhu. Learning career mobility and human activity patterns for job change analysis. In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 1057–1062, 2015.

[YCG16] Shipeng Yu, Evangelia Christakopoulou, and Abhishek Gupta. Identifying decision makers from professional social networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA*, pages 333–342, 2016.

[YDW+15] Wenchao Yu, Ariyam Das, Justin Wood, Wei Wang, Carlo Zaniolo, and Ping Luo. Max-intensity: Detecting competitive advertiser communities in sponsored search market. In *2015 IEEE International Conference on Data Mining (ICDM 2015), Atlantic City, NJ, USA*, pages 569–578, 2015.

[YYW+16] Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John C. S. Lui. Diversified temporal subgraph pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA*, pages 1965–1974, 2016.

[Zak99] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, October 1999.

[Zak00] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowl. and Data Eng.*, 12(3):372–390, May 2000.

[ZCF+16] Tianyang Zhang, Peng Cui, Christos Faloutsos, Yunfei Lu, Hao Ye, Wenwu Zhu, and Shiqiang Yang. Come-and-go patterns of group evolution: A dynamic model. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA*, pages 1355–1364, 2016.

[ZE05] Osmar R. Zaïane and Mohammad El-Hajj. Pattern lattice traversal by selective jumps. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD 2005), Chicago, USA*, pages 729–735. ACM, 2005.

[ZZG+13] Chuan Zhou, Peng Zhang, Jing Guo, Xingquan Zhu, and Li Guo. UBLF: an upper bound based approach to discover influential nodes in social networks. In *The 13th IEEE International Conference on Data Mining (ICDM 2013), Dallas, TX, USA*, pages 907–916. IEEE, 2013.