FREQUENT PATTERN SPACE MAINTENANCE:

THEORIES & ALGORITHMS

FENG MENGLING

SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

2009

FREQUENT PATTERN SPACE MAINTENANCE: THEORIES & ALGORITHMS

FENG MENGLING

SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

A thesis submitted to Nanyang Technological University

in fulfilment of the requirement for the Degree of

Doctor of Philosophy

2009

Acknowledgement

My supervisors, Limsoon Wong and Yap-Peng Tan, have been an inspiration throughout my Ph.D study. I would like to thank them for their patience, their encouragement and their guidance; they have greatly helped me throughout my study. I would like to thank Nanyang Technological University and National University of Singapore for providing me with both hardware and software support. I would like to thank A*STAR for providing me with a scholarship and an opportunity to pursue my Ph.D degree. I would also like to thank Guozhu Dong and Jinyan Li for their valuable advices and inspiring ideas. I am immensely grateful to JianFei Zhu, Mohammed Javeed Zaki, Bart Goethals and Yun Chi for their kindness in sharing their source codes with me. I am deeply grateful to all the staffs, technicians and students in both Nanyang Technological University and National University of Singapore for their help and support. Special thanks to Lian Naixiang, Guimei Liu, Haiquan Li and all my colleagues and friends in the Computational Biology Lab. Last but not least, I would like to thank my father and wife for their understanding, their kindness, their advices and their unceasing support and encouragement.

Abstract

This Thesis explores the theories and algorithms for frequent pattern space maintenance. Frequent pattern maintenance is essential for various data mining applications, ranging from database management to hypothetical query answering and interactive trend analysis. Through our survey, we observe that most existing maintenance algorithms are proposed as an extension of certain pattern discovery algorithms or the data structures they used. But, we believe that, to develop effective maintenance algorithms, it is necessary to understand how the space of frequent patterns evolves under the updates. We investigate the evolution of frequent pattern space using the concept of equivalence classes. This space evolution analysis lays a theoretical foundation for the construction of efficient algorithms. Based on the space evolution analysis, novel "maintainers" for the frequent pattern space, "Transaction Removal Update Maintainer" (TRUM) and "Pattern Space Maintainer" (PSM), are proposed. TRUM effectively addresses the decremental maintenance of frequent pattern space. PSM is a "complete maintainer" that effectively maintains the space of frequent patterns for incremental updates, decremental updates and support threshold adjustments. Experimental results demonstrate that both TRUM and PSM outperform the state-of-the-art discovery and maintenance algorithms by significant margins.

Contents

List of Figures

List of Tables

Part I

INTRODUCTION

Chapter 1

Introduction & Motivation

This Thesis discusses the theories and algorithms for the maintenance of frequent pattern space. “Frequent patterns”, also known as frequent itemsets, refer to patterns that appear frequently in a particular dataset [2]. Frequent patterns are defined based on a user-defined threshold, called the “support threshold”. Given a dataset, we say a pattern is a frequent pattern if and only if its occurrence frequency is above or equals to the support threshold. We also define the collection of all frequent patterns as the “frequent pattern space” or “the space of frequent patterns”. Frequent patterns are a very important type of patterns in data mining [32]. Frequent patterns play an essential role in various knowledge discovery tasks, such as the discovery of association rules [3], correlations [8], causality [65], sequential patterns [4], partial periodicity [30], emerging patterns [21], etc. In the last decade, the discovery of frequent patterns has attracted tremendous research attention, and a phenomenal number of discovery algorithms, such as [1, 2, 3, 35, 52, 56, 64, 75], are proposed.

The maintenance of the frequent pattern space is as crucial as the discovery of the pattern space. This is because data is dynamic in nature. Due to the advance in data generation and collection technologies, databases are constantly updated with newly collected data. Data updates are also used as a means in interactive data mining, to gauge the impact caused by hypothetical changes to the data and to detect emergence and disappearance of trends. When a database is often updated or modified for interactive mining, repeating the pattern discovery process from scratch causes significant computational and I/O overheads. Therefore, effective maintenance algorithms are needed to update and maintain the frequent pattern space. This Thesis focuses on the maintenance of frequent pattern space for *transactional datasets* [68].

We observe that most of the prior works in frequent pattern maintenance are proposed as an extension of certain frequent pattern discovery algorithms or the data structures they used. Unlike the prior works, this Thesis lays a theoretical foundation for the development of effective maintenance algorithms by analyzing the evolution of frequent pattern space in response to data changes. We study the evolution of pattern space using the concept of equivalence classes [20, 47]. Inspired by the evolution analysis, novel maintenance algorithms are proposed to handle various data updates.

1.1 Applications

Frequent pattern space maintenance is desirable for many data mining applications. We here list a few major ones.

Data update

The most obvious application of frequent pattern space maintenance is to address data updates. Data updates are a fundamental aspect of database management [13, 61]. Data updates allow new records to be inserted, obsolete records to be removed, and errors in the databases to be amended. Data updates directly cause the frequent pattern space to change, and thus efficient methods are needed to maintain the pattern space.

Support threshold adjustment

Frequent pattern space maintenance is also a useful tool for various interactive mining applications. Support threshold adjustment is one of the most common operations in interactive mining of frequent patterns. Setting the right support threshold is crucial in frequent pattern mining. An inadequate support threshold may produce too few patterns to be meaningful or too many to be processed. It is unlikely to set the appropriate threshold at the first time. Therefore, data analyzers often adjust the support threshold interactively to obtain the desirable set of patterns. Since the frequent pattern space is defined based on the support threshold, support threshold adjustments may invalidate existing frequent patterns or may induce new frequent patterns. Regenerating the frequent pattern space every time the support threshold is adjusted involves considerable amount of redundancy. In this case, pattern space maintenance methods can be employed to reduce redundancy and improve efficiency.

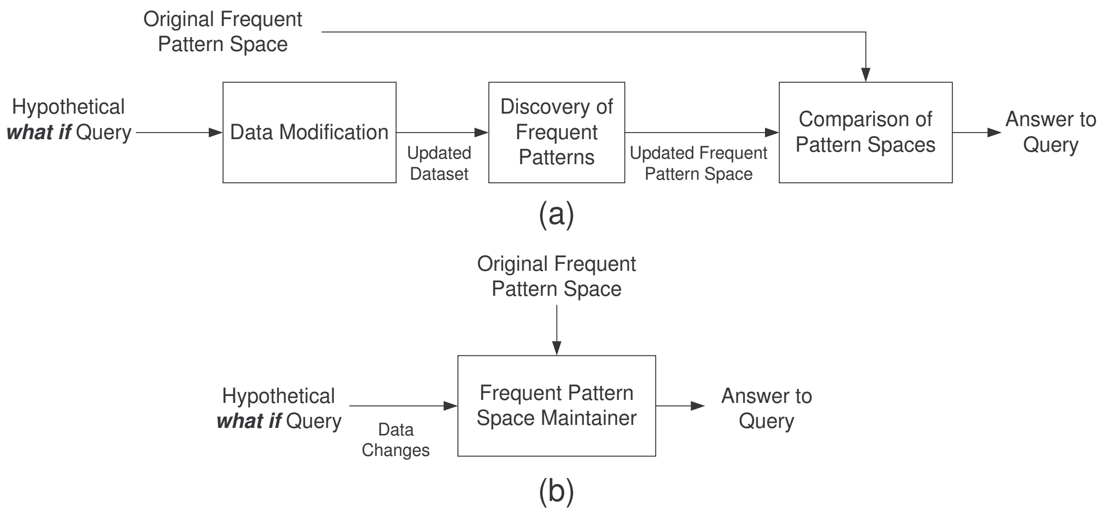

Original Frequent Pattern Space
Hypothetical *what if* Query
Data Modification
Updated Dataset
Discovery of Frequent Patterns
Updated Frequent Pattern Space
Comparison of Pattern Spaces
Answer to Query
(a)
Original Frequent Pattern Space
Hypothetical *what if* Query
Data Changes
Frequent Pattern Space Maintainer
Answer to Query
(b)

Figure 1.1: (a) The naive approach and (b) the pattern space maintenance approach for answering the hypothetical "*what if*" queries.

Hypothetical queries

Another potential application of frequent pattern maintenance in interactive mining is to answer hypothetical queries. In particular, the "*what if*" queries. Data analyzers are often interested to find out "*what*" might happen to the pattern space "*if*": some new transactions were inserted to the dataset, some existing transactions were removed, or a group of existing transactions were replaced with some new transactions, etc.

Given a dataset and the original frequent pattern space, the naive approach to answer the hypothetical *what if* queries is presented in Figure 1.1 (a). In the naive approach, one first needs to update the dataset according to the hypothetical query. Then, a frequent pattern mining algorithm is employed to generate the updated frequent pattern space based on the updated dataset. Finally, the original and updated frequent pattern spaces are compared to determine the impact of the hypothetical changes to the data. The naive approach is obviously inefficient. In general, the original and updated pattern spaces will be highly overlapped. Therefore, the naive approach involves a large amount of redundancy. Moreover, the size of frequent pattern spaces is usually very large [32], thus comparing the original and updated patterns is computationally expensive. Pattern space maintenance can be used to avoid these redundancy and inefficiency.

Figure 1.1 (b) illustrates how frequent pattern space maintenance can help to simplify the query answering process. The “Pattern Space Maintainer”, as shown in Figure 1.1 (b), takes both the exiting pattern space and hypothetical data changes as input. It then maintains the pattern space according to the data changes. As it is updating the pattern space, the maintainer at the same time detects the impacts of the hypothetical data changes. The pattern space maintainer can directly provide answers to hypothetical queries by reporting information such as how many existing patterns are affected by the hypothetical data changes, what these patterns are, which patterns have become infrequent, what patterns have newly emerged, etc. As a result, by employing pattern space maintenance, the pattern discovery and pattern space comparison steps in the naive approach are avoided.

Trend analysis

Trend analysis is another important application of pattern space maintenance. In trend analysis, different from the conventional frequent pattern applications, each transaction is associated with a time stamp, which can be the time, the date, the month or the year of the transaction.

To illustrate the details of trend analysis, let us use the well-known “supermarket

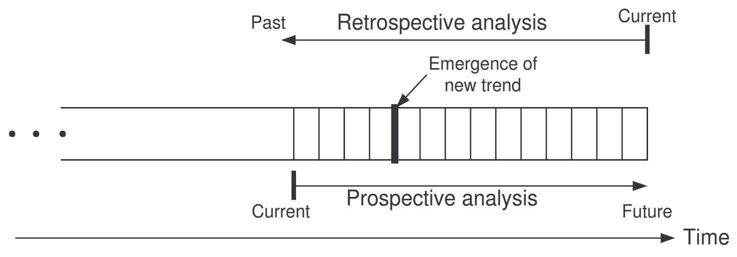


Figure 1.2: Retrospective and prospective trend analysis.

basket" case as an example. Suppose we have the transaction records of a supermarket, and each transaction record is associated with its date of purchase. Now the supermarket manager wants to study the buying trends of his customers. In this case, buying trends are actually equivalent to frequent patterns in the transactions. Thus emergence of new frequent patterns implies emergence of new buying trends, and disappearance of existing frequent patterns implies some existing trends have vanished.

Trend analysis can be both retrospective and prospective. In the retrospective scenario, the supermarket manager already knew that a new trend has emerged in the near past; but he/she does not know when the emergence happened. Suppose the manager would like to find out exactly when so that he/she can further investigate the causes of the new trend. In this case, the emergence of the new trend can be detected by comparing the current and past frequent pattern spaces as we slowly move backwards in time. Figure 1.2 graphically illustrates this retrospective process, and we call it the "retrospective trend analysis". On the other hand, in the prospective scenario, the manager has no idea whether any new trend will emerge, and he wants to know if there is any. In this case, the manager needs to update the data continually with the latest transactions and the compare the existing and updated frequent pattern space to find out whether any new trend has emerged. We call this case the "prospective trend analysis", and it is also demonstrated in Figure 1.2.

In real life applications, retrospective trend analysis is often employed to study customer buying trends and market trends as illustrated in the example above. Retrospective trend analysis is also applied to sensor data to detect faults and unusual readings. On the other hand, prospective trend analysis is then used to detect abnormal and unexpected behavior from databases like web log and money transactions. Although the example above only mentions the detection of new trends, trend analysis can also be employed to detect the disappearance of existing trends.

Trend analysis is a type of "before vs. after" analysis, which requires intensive pattern discovery and comparison computation. Pattern maintenance is the best option to avoid redundancy and thus to achieve high efficiency.

1.2 Types of Maintenance

This Thesis focuses on two major types of updates in data management and interactive mining:

The first type of update, where new transactions are inserted into the original dataset, is called an incremental update. The associated maintenance process is called incremental maintenance. Incremental maintenance is employed in Data-Base Management System (*DBMS*) to allow data users to update databases with new records. Incremental maintenance is also an effective tool for hypothetical tests to study how the pattern space may evolve if some hypothetical transactions are inserted.

In addition, as discussed in the previous section, incremental maintenance is necessary for "prospective trend analysis".

The second type of update, where some transactions are removed from the original dataset, is called a decremental update. The associated maintenance process is called decremental maintenance. Opposite of incremental maintenance, decremental maintenance is used in *DBMS* to remove obsolete and incorrect records; decremental maintenance is employed in hypothetical tests to investigate the "what if" effects on pattern space when some exiting records are hypothetically removed; and decremental maintenance can also be applied in "retrospective trend analysis".

Incremental and decremental maintenance of frequent pattern space can be done in two different manners: **eager maintenance** and **batch maintenance**. Take incremental maintenance as an example. Eager maintenance refers to the case where the dataset is updated constantly with new data, and the frequent pattern space needs to be updated as each new data arrives. Eager maintenance is suitable for applications that require instant updates of the pattern space, such as web monitoring, production line monitoring, etc. Eager maintenance is also required in the applications of trend analysis. On the other hand, batch maintenance refers to the case where new data comes in as a batch, and the frequent pattern space only needs to be updated when the whole batch of incremental data arrives. Batch maintenance is often applied in applications that are not as urgent, e.g. update of sales data. The proposed algorithms in this Thesis are able to effectively handle both eager maintenance and batch maintenance.

In addition, we also maintain the frequent pattern space for support threshold adjustment — an important operation of interactive mining.

1.3 Challenges in Maintenance

Data updates and support threshold adjustment may invalidate existing frequent patterns and induce new frequent patterns to emerge. As a result, the maintenance of frequent pattern space, in general, consists of two major computational tasks: one is to update existing patterns and the other is to generate newly emerged frequent patterns.

Without loss of generality, suppose the dataset is incrementally updated with m new records. The naive way to update the existing patterns is to scan through all the existing patterns for each new record to find out which patterns are affected by the record and then update the patterns accordingly. The computational complexity of this naive approach is $\mathcal{O}(N_{FP} \times m)$, where N_{FP} refers to the size of the existing pattern space and m refers to the number of new records. Frequent pattern spaces are usually huge, meaning that N_{FP} is very large. As a result, the naive approach is in general computationally too expensive to be practically feasible. Therefore, how to update the existing frequent patterns effectively is one of the major challenges in frequent pattern maintenance.

The other major challenge in frequent pattern maintenance is the generation of newly emerged frequent patterns. In theory, the number of possible candidates for the newly emerged frequent patterns equals to $(2^n - N_{FP})$, where n is the total number of attributes in the dataset. This implies that the search space for the new patterns is extremely sparse and unpleasantly large. Thus efficient techniques are required to generate newly emerged frequent patterns.

1.4 Contributions

This Thesis makes the following contributes in the field of frequent pattern space maintenance.

- A detailed survey on previously proposed frequent pattern maintenance algorithms is conducted. The existing algorithms are classified into four major categories, viz. *Apriori*-based, partition-based, prefix-tree-based and concise-representation-based algorithms. The advantages and limitations of these algorithms are investigated from both theoretical and experimental perspectives.

- A theoretical foundation for the development of effective frequent pattern maintenance algorithms is laid by analyzing the evolution of the frequent pattern space under data updates. The evolution of the frequent pattern space is studied using the concept of equivalence classes. It is demonstrated that equivalence classes of the updated frequent pattern space can be derived based on existing frequent equivalence classes and the data updates.

- Inspired by the space evolution analysis, an effective and exact algorithm — *Transaction Removal Update Maintainer* (TRUM) — is proposed to maintain the frequent pattern space for decremental updates. A novel data structure, *Transaction-ID Tree* (*TID-tree*), is developed to facilitate the decremental maintenance of frequent patterns. The effectiveness of TRUM is validated by experimental evaluations.

- A complete frequent pattern space maintainer, *Pattern Space Maintainer* (PSM), is proposed. PSM is made up of three maintenance components: PSM+, PSM- and PSM_{Δ}. PSM+ handles the incremental maintenance of frequent patterns, PSM- handles the decremental maintenance, and PSM_{Δ} addresses the maintenance of frequent pattern space for support threshold adjustment. All three components of PSM are developed based a new data structure, *Generator-Enumeration Tree* (*GE-tree*). *GE-tree* is a tree structure that allows compact storage of frequent pattern space and, more importantly, facilitates fast update of existing patterns and efficient generation of new frequent patterns.

1.5 Thesis Organization

This Thesis is composed of four parts: an introduction, a discussion on the theoretical foundation for effective frequent pattern space maintenance, introduction to our proposed maintenance algorithms, and a final conclusion with some future research directions.

Part I Introduction

Chapter 2 recaps the fundamental definitions of frequent pattern mining and formally defines the problem of frequent pattern space maintenance. Notations used in frequent pattern mining and maintenance are introduced. Basic properties and concise representations of frequent patterns are discussed. Related works in frequent pattern mining are also reviewed.

Chapter 3 surveys previously proposed maintenance algorithms. The previously proposed algorithms are classified into four categories based on their characteristics. The advantages and limitations of these four types of maintenance algorithms are investigated theoretically and experimentally.

Part II Theories

Chapter 4 analyzes how the frequent pattern space evolves under incremental updates, decremental updates and support threshold adjustments. The evolution of pattern space is studied based on the concept of equivalence classes. This evolution analysis lays the theoretical foundation for the development of effective maintenance algorithms.

Part III Algorithms

Chapter 5 introduces our proposed decremental maintenance algorithm — Transaction Removal Update Maintainer (TRUM). A novel data structure, *Transaction-ID Tree* (*TID-tree*), is proposed to optimize the efficient of TRUM. The construction and update of *TID-tree* is discussed. At the end of the chapter, the effectiveness of TRUM is evaluated with experimental studies, and the limitations and possible extensions of TRUM are also explored.

Chapter 6 proposes a complete frequent pattern space maintainer, named Pattern Space Maintainer (PSM). PSM consists of three maintenance components: PSM+, the incremental maintenance component, PSM-, the decremental maintenance component, and PSM_{Δ}, the support threshold maintenance component. All these three components are developed based on the newly introduced data structure, *Generator-Enumeration Tree* (*GE-tree*). The characteristics and advantages of *GE-tree* are first discussed. Then the three maintenance components are introduced. The effectiveness of the three maintenance components is also justified with experimental results.

Part IV Conclusion

Chapter 7 lists my publications during my Ph.D study and declares the relations between these publications and the results shown in the Thesis.

Chapter 8 summarizes the results of the Thesis. Some potential future research directions are also discussed.

Chapter 2

Background and Problem Definition

2.1 Background on Frequent Patterns

Let $\mathcal{I} = \{i_1, i_2, ..., i_m\}$ be a set of distinct literals called "items", and also let $\mathcal{D} = \{t_1, t_2, ..., t_n\}$ be a transactional "dataset", where t_i ($i \in [1, n]$) is a "transaction" that contains a non-empty set of items. Each *non-empty* subset of $\mathcal{I}$ is called a "pattern" or an "itemset". The "support" of a pattern P in a dataset $\mathcal{D}$ is defined as $sup(P, \mathcal{D}) = |\{t | t \in \mathcal{D} \wedge P \subseteq t\}|$. A pre-specified support threshold is necessary to define frequent patterns. The support threshold can be defined in terms of percentage and absolute count. For a dataset $\mathcal{D}$, the "percentage support threshold", $ms_{\%}$, and the "absolute support threshold", ms_a, can be interchanged via equation $ms_a = \lceil ms_{\%} \times |\mathcal{D}| \rceil$. Given $ms_{\%}$ or ms_a, a pattern P is said to be *frequent* in a dataset $\mathcal{D}$ iff $sup(P, \mathcal{D}) \geq ms_a = \lceil ms_{\%} \times |\mathcal{D}| \rceil$. When $ms_{\%}$ is used, for simplicity of discussion, we

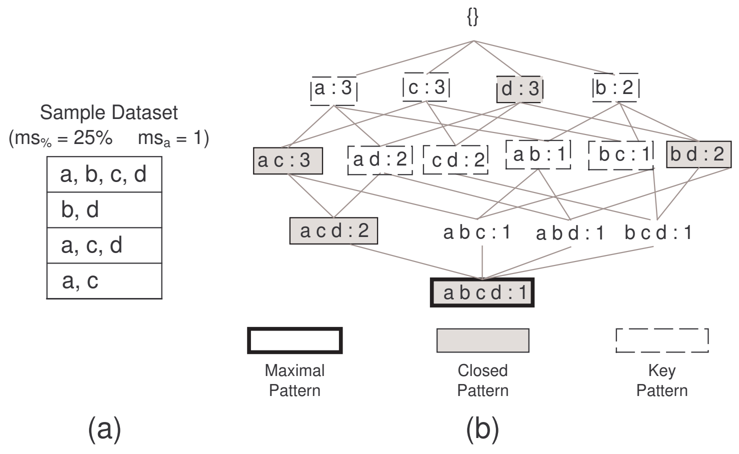


Figure 2.1: (a) An example of transactional dataset, which has 4 transactions, and (b) the frequent pattern space for the sample dataset in (a) and its concise representations ($ms_{\%} = 25\%$, $ms_a = 1$).

say the support of a pattern P is greater or equals to $ms_{\%}$ if $sup(P, \mathcal{D}) \geq \lceil ms_{\%} \times |\mathcal{D}| \rceil$. The collection of all frequent patterns in $\mathcal{D}$ is called the "space of frequent patterns" or the "frequent pattern space" and is denoted as $\mathcal{F}(\mathcal{D}, ms_{\%})$ or $\mathcal{F}(\mathcal{D}, ms_a)$. Figure 2.1 shows an example of a transactional dataset and the frequent pattern space of the dataset when $ms_{\%} = 25\%$ ($ms_a = 1$).

The most important property of frequent patterns is the "*a priori*" property, which is also known as the "anti-monotonicity" property. The "*a priori*" property not only guides the discovery of frequent patterns, and it also inspires the concise representations of the pattern space.

Fact 2.1 (*A priori* Property [32]). *All non-empty subsets of a frequent pattern are also frequent, and every superset of an infrequent pattern is also infrequent.*

2.1.1 Concise Representation of Frequent Patterns

Frequent pattern space is usually very large. Moreover, the space grows exponentially as the support threshold drops. Take the *mushroom* dataset [25] as an example. The dataset consists of about 8 thousand transactions and about 100 distinct items. The corresponding frequent pattern space already contains almost 600 thousands patterns when the support threshold $ms_{\%} = 10\%$, and the space grows to 90 million patterns when the support threshold drops to 1%. Therefore, concise representations [7, 10, 35, 38, 49, 58] are proposed to summarize the frequent pattern space.

The commonly used concise representations of frequent patterns are the **maximal patterns** [35], **closed patterns** [57] and **generators** (also know as key patterns) [57]. Figure 2.1 (b) graphically illustrates how the concise representations can be used to summarize the frequent pattern space.

Maximal patterns are first introduced in [35]. Maximal frequent patterns (in short maximal patterns) refer to the longest patterns that are frequent, and they are formally defined as follows.

Definition 2.2 (Maximal Pattern)**.** *Given a dataset $\mathcal{D}$ and the minimum support threshold $ms_{\%}$, a pattern P is a "maximal frequent pattern" iff $sup(P, \mathcal{D}) \geq ms_{\%}$ and, for every $Q \supset P$, it is the case that $sup(Q, \mathcal{D}) < ms_{\%}$.*

Maximal patterns are the most compact representation of frequent pattern space. As shown in Figure 2.1 (b), a single maximal pattern is already sufficient to represent the entire pattern space, which consists of 15 patterns in total. Maximal patterns can be used to enumerate the complete set of frequent patterns. According to the *a priori* property, all subsets of maximal patterns are frequent. However, maximal patterns lack the information to derive the exact support of all frequent patterns. Therefore, maximal patterns are a lossy[1] representation.

Closed patterns and generators are both defined by N. Pasquier et al. in [57]. Unlike maximal patterns, the closed pattern and generator representations are lossless[1] concise representations of frequent pattern space.

Definition 2.3 (Closed Pattern & Generator)**.** *Given a dataset $\mathcal{D}$, a pattern P is a "closed pattern" iff for every $Q \supset P$, it is the case that $sup(Q, \mathcal{D}) < sup(P, \mathcal{D})$. A pattern P is a "generator" or a "key pattern" iff for every $Q \subset P$, it is the case that $sup(Q, \mathcal{D}) > sup(P, \mathcal{D})$.*

Closed patterns are the most specific patterns that have a particular support. The closed pattern representation is composed of the set of frequent closed patterns annotated with their support values. Given a dataset $\mathcal{D}$ and a minimum support threshold $ms_{\%}$, we denote the closed pattern representation as $\mathcal{FC}(\mathcal{D}, ms_{\%})$. As can be observed from Figure 2.1 (b), closed pattern representation is not as compact as maximal patterns. However, the representation is lossless. The representation can be used to generate all frequent patterns based on the *a priori* property, and it can also be used to derive the exact support of all frequent patterns. For any frequent pattern P in dataset $\mathcal{D}$, $sup(P, \mathcal{D}) = \max\{sup(C, \mathcal{D}) | C \supseteq P, C \in \mathcal{FC}(\mathcal{D}, ms_{\%})\}$.

Generators, on the other hand, are the most general patterns that have a particular support. The generator representation is made up with the set of frequent

[1]We say a concise representation of frequent pattern space is lossless if it is sufficient to derive the complete set of frequent patterns and their exact support values; and we say a representation is lossy if otherwise.

generators and their support values. Given a dataset $\mathcal{D}$ and a minimum support threshold $ms_{\%}$, we denote the generator representation as $\mathcal{FG}(\mathcal{D}, ms_{\%})$. Generator representation is also lossless. For any frequent pattern P in dataset $\mathcal{D}$, $sup(P, \mathcal{D}) = \min\{sup(G, \mathcal{D}) | G \subseteq P, G \in \mathcal{FG}(\mathcal{D}, ms_{\%})\}$.

Besides the maximal patterns, closed patterns and generators, other frequent pattern concise representations include the *free-sets* [7], *disjunction-free generators* [38] and *positive border patterns* [49]. However, these three representations are rarely used in frequent pattern applications. This is because they all involve complicated procedures and calculations when inferring the support values of frequent patterns. As a result, we will not discuss them in detail in this Thesis.

2.1.2 Discovery of Frequent Patterns

The discovery of frequent patterns in transactional datasets has been studied popularly in data mining research. In this section, we discuss the representative algorithms.

Apriori-based algorithms

Apriori [3] is the most influential algorithm for frequent pattern discovery. Many discovery algorithms [1, 36, 41, 52, 54, 56, 63] are inspired by Apriori. Apriori employs a "candidate-generation-verification" framework. The algorithm generates its candidate patterns using a "level-wise" search. The essential idea of the level-wise search is to iteratively enumerate the set of candidate patterns of length $(k+1)$ from the set of frequent patterns of length k. The support of candidate patterns will then be counted by scanning the dataset.

One major drawback of Apriori is that it leads to the enumeration of a huge number of candidate patterns. For example, if a dataset has 100 items, Apriori may need to generate $2^{100} \approx 10^{30}$ candidates. Another drawback of Apriori is that it requires multiple scans of the dataset to count the support of candidate patterns. Different variations of Apriori are proposed to address these limitations. E.g. J. S. Park et al. introduced a hash-based technique in [56] to reduce the size of candidate patterns. J. Han et al. [31] proposed to speed up the support counting process by reducing the number of transactions scanned in future iterations. The idea of [31] is that a transaction that does not contain any frequent pattern of length k cannot contain any frequent pattern with length greater than k. Therefore, such transactions can be ignored for subsequent iterations.

Partition-based algorithms

Partition-based algorithms are another type of discovery algorithms. Similar to Apriori, partition-based algorithms follow the candidate-generation-verification framework. However, partition-based algorithms generate candidate patterns in a different manner. The partitioning candidate generation technique is first introduced in [64]. Given a dataset $\mathcal{D}$ and a support threshold $ms_{\%}$, the candidate patterns are generated in two phases. In phase I, the dataset $\mathcal{D}$ is divided into n partitions $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_n$, where $\mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_n = \mathcal{D}$ and $\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$ for $1 \leq i < j \leq n$. The frequent patterns of each partition are first generated. These patterns are referred as "local frequent patterns". In phase II, the dataset is scanned to determine "global frequent patterns" from the set of local frequent patterns. Partition-based algorithms are developed based on the fact that: any global frequent pattern must be locally frequent in at least one of the partitions.

Both Apriori-based and partition-based algorithms employ the candidate-generation-verification framework. Their computational complexity is proportionally related to the number of enumerated candidate patterns. Although various pruning techniques [1, 31, 56] have been proposed to reduce the number of candidate patterns, a large number of unnecessary candidate patterns are still enumerated. This greatly reduces the performance of both Apriori-based and partition-based algorithms.

Prefix-tree-based algorithms

To address the shortcoming of the candidate-generation-verification framework, prefix-tree-based algorithms, which involve no candidate generation, are proposed. Examples of prefix-tree-based algorithms include [28, 33, 50, 59].

FP-growth described in [33] is the state-of-the-art prefix-tree-based discovery algorithm. FP-growth mines frequent patterns based on a prefix-tree structure, *Frequent Pattern Tree* (*FP-tree*).

FP-tree is a compact representation of all relevant frequency information in a database. Every branch of the *FP-tree* represents a "projected transaction" and also a candidate pattern. The nodes along the branches are stored in descending order of the support values of corresponding items, so leaves are representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping transactions share prefixes of the corresponding branches. Figure 2.2 demonstrates how *FP-tree* is constructed for the sample dataset given a support threshold $ms_{\%}$. First, the dataset is transformed into the "projected dataset". In the "projected

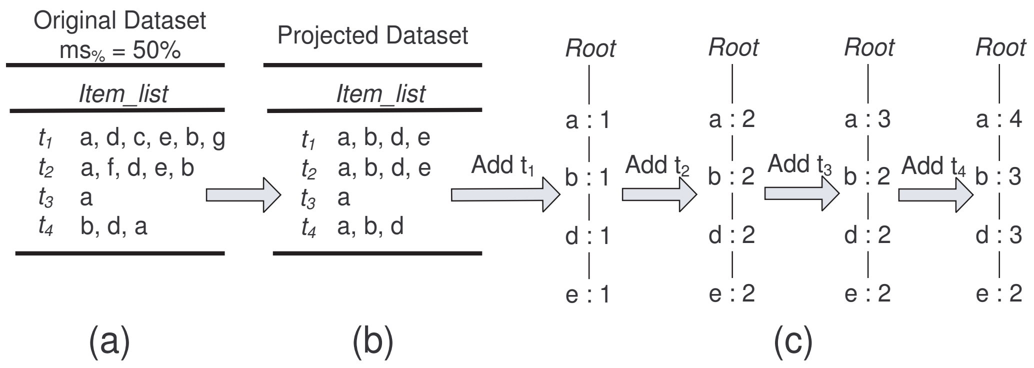


Figure 2.2: (a) The original dataset, (b) the projected dataset when $ms_{\%} = 50\%$ and (c) the construction process of *FP-tree*.

dataset", all the infrequent items are removed, and items in each transaction are sorted in descending order of their support values. Transactions in the "projected dataset" are named the "projected transactions". The "projected transactions" are then inserted into the prefix-tree structure one by one, as shown in Figure 2.2 (c). It can be seen that the *FP-tree* effectively represents the sample dataset with only four nodes.

With *FP-tree*, FP-growth generates frequent patterns using a "fragment growth technique". The fragment growth technique enumerates frequent patterns based on the support information stored in *FP-tree*, which effectively avoids the generation of unnecessary candidate patterns. Inspired by the idea of divide-and-conquer, the fragment growth technique decomposes the mining tasks into subtasks that mines frequent patterns for conditional datasets, which greatly reduces the search space. Details of the technique can be referred to [33].

FP-growth significantly outperforms both the Apriori-based and partition-based algorithms. The advantages of FP-growth are: first, *FP-tree* effectively compresses and summarizes the dataset so that multiple scans of dataset is no longer needed to obtain the support of patterns; second, the fragment growth technique ensures no unnecessary candidate patterns are enumerated; lastly, the search task is simplified with a divide-and-conquer method. However, FP-growth, like other prefix-tree based algorithms, still suffers from the undesirable large size of the frequent pattern space. To break this bottleneck, algorithms are proposed to discover the concise representations of frequent pattern space.

Concise-representation-based algorithms

Instead of mining the whole frequent pattern space, concise-representation-based algorithms aim to generate only the concise representation patterns, such as maximal patterns, closed patterns and generators.

In the literature, algorithms Max-miner [35], MAFIA [9], FPmax* [28] and GENMAX [26] are proposed to discover maximal patterns. Among these algorithms, FPmax* is, in general, the fastest algorithm [25]. FPmax* is an extension of FP-growth, and FPmax* is also developed based on the *FP-tree* structure. Extra pruning techniques are used in FPmax* to remove non-maximal patterns in the early stage of mining.

Closed patterns are the most popular concise representation in the literature. A large number of algorithms are proposed to mine closed patterns. Algorithms A-close [57], CHARM [74], CLOSET [60], CLOSET+ [72], LCM [69] and FPclose [28] are the representative ones. Most of the closed pattern mining algorithms made use of the following two properties of closed patterns to prune away non-closed patterns.

Fact 2.4 (Cf. [53]). *Given a dataset $\mathcal{D}$ and pattern C as a closed pattern. A pattern*

P is definitely not a closed pattern if $P \subset C$ and $sup(P, \mathcal{D}) = sup(C, \mathcal{D})$.

Fact 2.5 (Cf. [53]). *Let the filter, $f(P, \mathcal{D})$, of a pattern P in dataset $\mathcal{D}$ be defined as $f(P, \mathcal{D}) = \{t \in \mathcal{D} | P \subseteq t\}$. For any two patterns P and Q, if $f(P, \mathcal{D}) \subseteq f(Q, \mathcal{D})$ and $P \not\supset Q$, then P and all P's supersets are not closed patterns.*

Generators did not receive much attention from data mining research [48]. Very few works have discussed the discovery of generators. Some algorithms, e.g. A-close and GC-growth [47], discover frequent generators as they enumerate closed patterns.

2.2 Problem Definition

This section formally defines the problem of frequent pattern space maintenance.

For incremental maintenance, we use the following notations: $\mathcal{D}_{org}$ is the original dataset, $\mathcal{D}_{inc}$ is the set of new transactions to be added to $\mathcal{D}_{org}$, and $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \mathcal{D}_{inc}$ is the updated dataset. We assume without loss of generality that $\mathcal{D}_{org} \neq \emptyset$ and $\mathcal{D}_{org} \cap \mathcal{D}_{inc} = \emptyset$. This leads to the conclusion that $|\mathcal{D}_{upd+}| = |\mathcal{D}_{org}| + |\mathcal{D}_{inc}|$. Given $ms_{\%}$, the task of incremental maintenance is to obtain the updated frequent pattern space $\mathcal{F}(\mathcal{D}_{upd+}, ms_{\%})$ by updating the original pattern space $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$. Note that, in this Thesis, we focus on maintenance methods that generate the **complete** set of frequent patterns along with their **exact** support values.

Analogously, we use the following notations for decremental maintenance: $\mathcal{D}_{dec}$ is the set of old transactions to be removed, and $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \mathcal{D}_{dec}$ is the updated dataset. We assume without loss of generality that $\mathcal{D}_{org} \neq \emptyset$ and $\mathcal{D}_{dec} \subseteq \mathcal{D}_{org}$. Thus

$|\mathcal{D}_{upd-}| = |\mathcal{D}_{org}| - |\mathcal{D}_{dec}|$. Given $ms_{\%}$, the task of decremental maintenance is to obtain the updated frequent pattern space $\mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$ by updating the original pattern space $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$.

For the scenario of support threshold adjustment, let ms_{org} be the original minimum support threshold and ms_{upd} be the updated support threshold. The maintenance task is to obtain the updated frequent pattern space $\mathcal{F}(\mathcal{D}, ms_{upd})$ based on the original pattern space $\mathcal{F}(\mathcal{D}, ms_{org})$.

Chapter 3

Existing Maintenance Algorithms: A Survey

This section surveys previously proposed frequent pattern maintenance algorithms. We investigate the advantages and limitations of existing maintenance algorithms from both algorithmic and experimental perspectives. We observe that most existing algorithms are proposed as an extension of certain frequent pattern discovery algorithms or the data structure they used. Therefore, same as frequent pattern discovery algorithms, existing maintenance algorithms can be mainly classified into 4 categories: *Apriori-based*, *Partition-based*, *Prefix-tree-based* and *Concise-representation-based* algorithms. Representative algorithms for all 4 categories are discussed. This chapter should not be taken as an exhaustive account as there are too many existing approaches to be included.

3.1 Algorithmic Studies

We first study the mechanisms of existing algorithms from an algorithmic point of view. The pros and cons of existing algorithms are theoretically compared and illustrated.

3.1.1 Apriori-based Algorithms

Apriori-based algorithms [6, 14, 15, 77], as the name suggests, are inspired by the well-known frequent pattern discovery algorithm, Apriori [3].

FUP [14] is the representative Apriori-based maintenance algorithm. It is proposed to address the incremental maintenance of frequent patterns. FUP updates the space of frequent patterns based on the candidate-generation-verification framework of Apriori. Different from Apriori, FUP makes use of the support information of the previously discovered frequent patterns to reduce the number of candidate patterns. FUP effectively prunes unnecessary candidate patterns based on the following two observations.

Fact 3.1. *Let $\mathcal{D}_{org}$ be the original dataset, $\mathcal{D}_{inc}$ be the incremental dataset, $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \mathcal{D}_{inc}$ be the updated dataset and $ms_{\%}$ be the support threshold. Also let P be a frequent pattern in $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$. For every $Q \supset P$, $Q \notin \mathcal{F}(\mathcal{D}_{upd+}, ms_{\%})$, if $P \notin \mathcal{F}(\mathcal{D}_{upd+}, ms_{\%})$.*

Fact 3.1 is an extension of the *a priori* property. It is to say that, if a previously frequent pattern becomes infrequent in the updated dataset, then all its supersets are definitely infrequent in the updated dataset and thus should not be included as candidate patterns. FACT 3.1 facilitates us to discard existing frequent patterns that are no longer frequent. FACT 3.2 then provides us a guideline to eliminate unnecessary candidates for newly emerged frequent patterns.

Fact 3.2. *Let $\mathcal{D}_{org}$ be the original dataset, $\mathcal{D}_{inc}$ be the incremental dataset, $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \mathcal{D}_{inc}$ be the updated dataset and $ms_{\%}$ be the support threshold. For any pattern P, $sup(P, \mathcal{D}_{upd+}) < ms_{\%}$ and $P \notin \mathcal{F}(\mathcal{D}_{upd+}, ms_{\%})$, if $sup(P, \mathcal{D}_{org}) < ms_{\%}$ and $sup(P, \mathcal{D}_{inc}) < ms_{\%}$.*

Fact 3.2 states that, if a pattern is infrequent in both the original dataset and the incremental dataset, it is definitely infrequent in the updated dataset. This allows us to eliminate unqualified candidates of newly emerged frequent patterns based on their support values in the incremental dataset. These support values can be obtained by scanning only the incremental dataset, which is in general much smaller. This greatly reduces the number of scans of the original dataset and thus improves the effectiveness of the algorithm.

FUP is generalized in [15] to also maintain the frequent pattern space for decremental updates and support threshold adjustments. The generalized version of FUP is called FUP2H. Both FUP and FUP2H generate a much smaller set of candidate patterns compared to Apriori, and thus they are more effective. But both FUP and FUP2H still suffer from two major drawbacks: (1) they require multiple scans of the original and incremental/decremental datasets to obtain the support values of candidate patterns, which leads to high I/O overheads; and (2) they repeat the enumeration of previously discovered frequent patterns. To address Point (2), Aumann et al [5] proposed a new algorithm — Borders.

Borders is inspired by the concept of the “border pattern”, introduced by Mannila and Toivonen [51]. In the context of frequent patterns, the “border pattern” is formally defined as follows.

Definition 3.3 (Border Pattern)**.** *Given a dataset $\mathcal{D}$ and minimum support threshold $ms_{\%}$, a pattern P is “border pattern” iff for every $Q \subset P$, $Q \in \mathcal{F}(\mathcal{D}, ms_{\%})$ but $P \notin \mathcal{F}(\mathcal{D}, ms_{\%})$.*

The border patterns are basically the shortest infrequent patterns. The collection of border patterns defines a borderline between the frequent patterns and the infrequent ones. Different from FUP, Borders makes use of not only the support information of previously discovered patterns but also the support information of the border patterns.

When the original dataset $\mathcal{D}_{org}$ is updated with incremental dataset $\mathcal{D}_{inc}$, Borders first updates the support values of existing frequent patterns and border patterns. If no border patterns emerge to be frequent after the update, the maintenance process is finished. Otherwise, if some border patterns become frequent after the update, new frequent patterns need to be enumerated. Border patterns that emerge to be frequent after the update are called the ”promoted border patterns”. The new pattern enumeration process follows the Apriori candidate-generation-verification method. But, distinct from Apriori and FUP, Borders resumes the pattern enumeration from the “promoted border patterns” onwards and thus avoids the enumeration of previously discovered frequent patterns.

Since Borders successfully avoids unnecessary enumeration of previously discovered patters, it is more effective than FUP. However, similar to FUP, Borders requires multiple scans of original and incremental/decremental datasets to obtain the support values of newly emerged frequent pattern and border patterns. Thus Borders still suffers from heavy I/O overheads.

On the other hand, to avoid unnecessary updates, Lee et al [40] extended the idea of FUP2H with sampling techniques and proposed a new algorithm — DELI. FUP2H obtains the exact support values of frequent patterns through multiple scans of the original and incremental/decremental datasets. Different from FUP2H, DELI estimates the support values of frequent patterns through sampling the datasets. Based on the estimated support values, DELI then calculates an approximate upper bound on how much change is introduced by the data updates to the existing frequent pattern space. “If the change is not significant, we can ignore it and wait until more is accumulated, contenting with the old” pattern space “as a good approximation.” [40] Otherwise, FUP2H is applied to update the frequent pattern space.

With sampling techniques and statistical approximations, DELI avoids unnecessary scans of datasets and trivial updates of existing patterns. As a result, compared with FUP2H, DELI is more effective, especially when data updates have introduced negligible changes to the existing pattern space. However, the performance of DELI greatly depends on the sampling size and the error tolerance level. If the error tolerance level is zero, DELI will basically perform like FUP2H. Moreover, DELI only generates a set of frequent patterns that “approximate” the actual frequent pattern space, and DELI can only obtain the “estimated” support values of frequent patterns. As mentioned, this Thesis focuses on algorithms that generate the “complete” set of frequent patterns along with their "exact" support values. Therefore, DELI is not our focus and is not included in our experimental studies.

3.1.2 Partition-based Algorithms

Partition-based maintenance algorithms [12, 39, 55] are developed based on the "partitioning heuristic", which is first described in [64]. Given a dataset $\mathcal{D}$ that is divided into n non-overlapping partitions $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_n$, the "partitioning heuristic" states that if a pattern P is frequent in $\mathcal{D}$ then P must be frequent in at lease one of the n partitions of $\mathcal{D}$.

Sliding Window Filtering (SWF) [39] is a recently proposed partition-based maintenance algorithm. SWF focuses on the pattern maintenance of time-variant datasets. In time-variant datasets, data updates involve both the insertion of the most recent transactions (incremental update) and the deletion of the most obsolete transactions (decremental update).

Given a time-variant dataset $\mathcal{D}$, similar to partition-based frequent pattern discovery algorithms, SWF first generates the locally frequent patterns for each partition as candidate patterns and then scan the entire dataset to determine which candidate patterns are globally frequent. For incremental maintenance, SWF treats the incremental dataset $\mathcal{D}_{inc}$ as a new partition, and the locally frequent patterns in $\mathcal{D}_{inc}$ are included as candidate patterns. For decremental maintenance, to facilitate the update of candidate patterns, SWF records not only the support information but also the "start partition" of each candidate pattern. The "start partition" of a candidate pattern refers to the first partition that the candidate pattern is first introduced. When the obsolete transactions (or partitions) are removed, the "start partition" attribute allows us to easily determine and update the candidate patterns that are involved in the removed transactions (or partitions).

The major advantage of SWF is that, based on the "partitioning heuristic", SWF prunes most of the false candidate patterns in the early stage of the maintenance process. This greatly reduces the computational and memory overhead. Moreover, SWF requires only one scan of the entire time-variant dataset to verify the set of candidate patterns. However, since SWF also follows the "candidate-generation-verification framework", even with all the candidate pruning techniques, SWF still produces a large number of unnecessary candidates. This drawback is the bottleneck for all candidate-generation-verification based algorithms.

3.1.3 Prefix-tree-based Algorithms

We have described in Section 2.1.2 how prefix-tree structures, e.g. the *FP-tree*, can be used to mine the frequent pattern space effectively when the dataset is static. In this section, we discuss how prefix-tree structures can be applied to maintain the frequent pattern space when the dataset is updated.

Koh and Shieh [37] developed an algorithm, AFPIM (*Adjusting FP-tree for Incremental Mining*), to update the *FP-tree* and frequent pattern space when the dataset is incrementally updated. AFPIM aims to update the previously constructed *FP-tree* by scanning only the incremental dataset. Recall that, in *FP-tree*, frequent items are

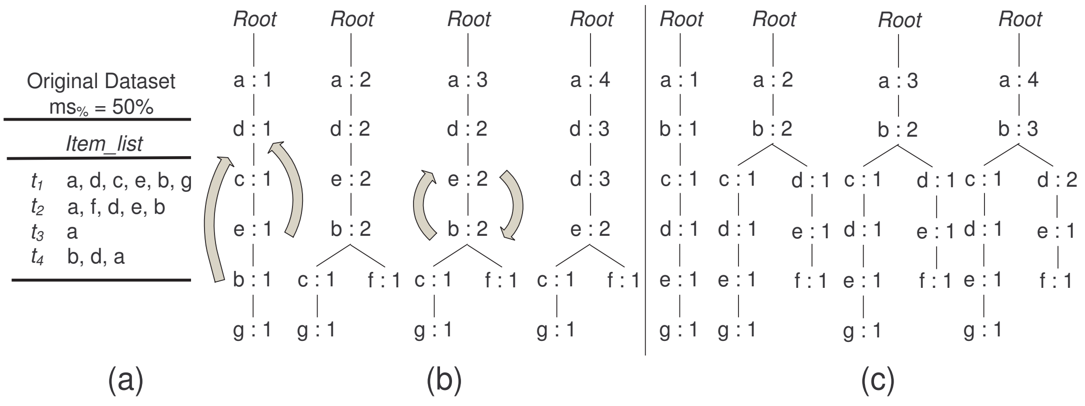


Figure 3.1: (a) The original dataset, (b) the construction process of *CATS-tree* and (c) the construction process of *CanTree*, where, in this case, items are sorted in lexicographic order.

arranged in descending order of their support values. Insertion of transactions may affect the support values and thus the ordering of items in the *FP-tree*. Therfore, when the ordering is changed, items in the *FP-tree* need to be adjusted. In AFPIM, this adjustment is accomplished by re-sorting the items through "bubble sort". Bubble sort involves recursively exchanges of adjacent items and thus is computational expensive, especially when the ordering of items are dramatically affected by the data updates. In addition, incremental update may induce new frequent items to emerge. In this case, the *FP-tree* can no longer be adjusted using AFPIM. AFPIM then has to scan the updated dataset to construct a new *FP-tree*.

CATS tree (Compressed and Arranged Transaction Sequences tree), a novel prefix-tree proposed by Cheung and Zaïane [16], on the other hand, does not suffer from the limitations of AFPIM. The *CATS tree* introduces a few new features. First, the *CATS tree* stores all the items in the transactions, regardless whether the items are frequent or not. This feature of *CATS tree* allows us to update *CATS tree* even when new frequent items have emerged. Second, to achieve high compactness, *CATS tree* arranges nodes based on their local support values. Figure 3.1 (b) illustrates how the *CATS tree* of the dataset in Figure 3.1 (a) is constructed and how the nodes in the tree are locally sorted. In the case of incremental updates, the *CATS tree* is updated by merging the new transactions one by one into the existing tree branches. This requires traversing the entire *CATS tree* to find the right path for the new transaction to merge in. In addition, since nodes in CATS tree are locally sorted, swapping and merging of nodes, as shown in Figure 3.1 (b), are required during the update of the *CATS tree*.

CanTree [43, 44], Canonical-order Tree, is another prefix-tree designed for the maintenance of frequent patterns. The *CanTree* is constructed in a similar manner as the *CATS tree*, as shown in Figure 3.1 (c). But in *CanTree*, items are arranged according to some canonical order, which is determined by the user prior to the mining process. For example, items can be arranged in lexicographic order, or, alternatively, items can be arranged based on certain property values of items (e.g. their prices, their priority values, etc.). Note that, in *CanTree*, once the ordering of items is fixed, items will follow this ordering for all the subsequent updates. To handle data updates, *CanTree* allows new transactions to be inserted easily. Unlike *CATS tree*, transaction insertions in *CanTree* require no extensive searching for merge-able paths. Also since the canonical order is fixed, any changes in the support values of items caused by data updates have no effect on the ordering of items. As a result, swapping/merging nodes are not required in the update of *CanTree*. The simplicity of *CanTree* makes it a very powerful prefix-tree structure for frequent pattern maintenance. Therefore, in our experimental studies, we choose algorithm CanTree to represent the prefix-tree-based maintenance algorithms.

In summary, the advantages of *Prefix-tree-based* maintenance algorithms are: (1) they concisely summarize the datasets into a prefix tree, which can be constructed with only two data scans; and (2) they effectively updates the frequent pattern space without enumeration of false candidates. On the other hand, the major drawback of the *Prefix-tree-based* algorithms is: they aim to maintain the entire frequent pattern space, which is huge in general. To address this drawback, algorithms are proposed to maintain only the concise representations of frequent pattern space.

3.1.4 Concise-representation-based Algorithms

Algorithms are proposed to maintain the concise representations of frequent pattern space. ZIGZAG [70] and moment [17, 18] are representative examples.

ZIGZAG maintains the maximal frequent patterns. ZIGZAG updates the maximal frequent patterns with a backtracking search, which is guided by the outcomes of the previous mining iterations. The backtracking search method in ZIGZAG is inspired by its related work GENMAX [27]. ZIGZAG conceptually enumerates the candidates of maximal frequent patterns with a "backtracking tree". Figure 3.2 (b) shows an example of backtracking tree. In the backtracking tree, each node is associated with a frequent pattern and its "combination set". For a particular frequent pattern P, the "combination set" refers to the set of items that form potential candidates by combining with P. Take the backtracking tree in Figure 3.2 (b) as an example. Node $\{a\}$ is associated with combination set $\{b, c, d\}$. This implies that the union of $\{a\}$

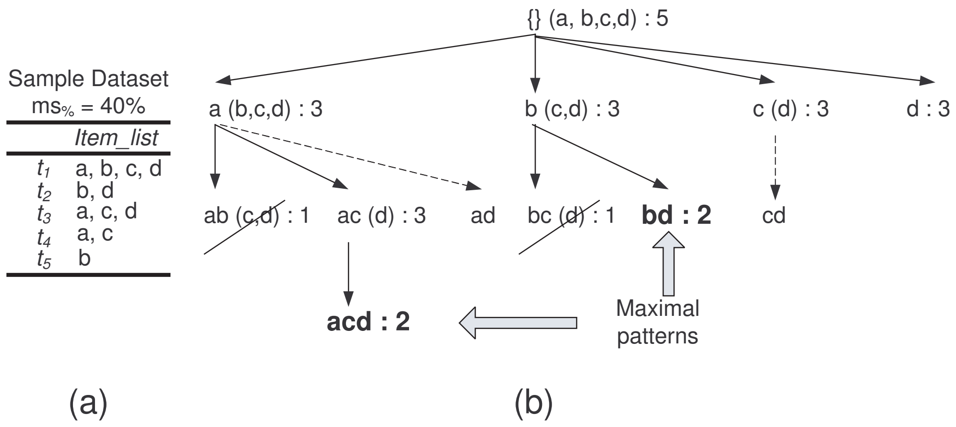


Figure 3.2: (a) Sample dataset and (b) the backtracking tree of the sample dataset when $ms_{\%} = 40\%$. In (b), bolded nodes are the frequent maximal patterns, nodes that are crossed out are enumeration termination points, and nodes that are linked with dotted arrows are skipped candidates.

and the items in the combination set, which are $\{a, b\}$, $\{a, c\}$ and $\{a, d\}$, are potential candidates for maximal frequent patterns.

ZIGZAG also employs certain pruning techniques to reduce the number of generated false candidates. First, ZIGZAG prunes false candidates based on the *a priori* property of frequent patterns. If a node in the backtracking tree is not frequent, then all the children of the node are not frequent, and thus candidate enumeration of the current branch can be terminated. In Figure 3.2 (b), crossed out nodes are the enumeration termination points that fall in this scenario. Second, ZIGZAG further eliminates false candidates based on the following fact.

Fact 3.4. *Given a dataset $\mathcal{D}$ and the support threshold $ms_{\%}$, let P be a maximal frequent pattern. It is true for every $Q \subset P$ that Q is not a maximal frequent pattern. It is also true for every $Q' \supset P$ that Q' is not a maximal frequent pattern.*

Fact 3.4 follows the definition of the maximal frequent pattern. In Figure 3.2 (b), nodes, which are pointed with a dotted line, are those pruned based on this criterion.

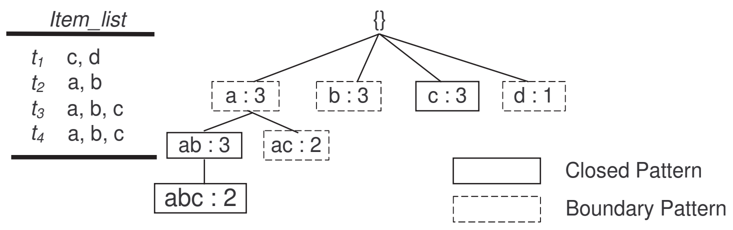


Figure 3.3: An example of *Closed Enumeration Tree* (*CET*). Patterns in solid line boxes are closed patterns, and patterns in dotted line boxes are boundary patterns.

Algorithm moment, on the other hand, maintains the frequent closed patterns. Algorithm moment focuses on the scenario when the dataset is updated in a sliding window manner, where, at each update, only one new transaction is inserted and only one existing transaction is removed. Algorithm moment is developed based on the hypothesis that there are only small changes to the frequent closed patterns in sliding window updates, because there is only a small amount of updates.

Algorithm moment employs a novel compact data structure, the *Closed Enumeration Tree* (*CET*), to facilitate the maintenance process. *CET* is a tree structure that compactly stores the frequent closed patterns and "boundary patterns". Figure 3.3 shows an example of *CET*. In Figure 3.3, patterns in solid boxes are frequent closed patterns, and patterns in dotted line boxes are boundary patterns. In moment, boundary patterns refer to patterns that are likely to emerge as frequent closed patterns after the update. According to the hypothesis of moment, the set of boundary patterns is believed to be relatively stable and requires little update. In addition, moment also applies rules in Facts 2.4 and 2.5 to prune non-closed candidates from *CET*.

It is demonstrated in [17, 18] that moment outperforms the frequent closed pattern discovery algorithm, *CHARM*, by multiple orders of magnitude. Moreover, moment can be generalized to handle batch updates. However, when the size of update gets large, the hypothesis of moment no longer holds, and therefore the performance of moment degrades dramatically.

3.2 Experimental Studies

In this section, we justify our theoretical observations with empirical results. Experiments were run on a PC with 2.4GHz CPU and 3.2G of memory. The performance of previously proposed maintenance algorithms is tested with a group of benchmark datasets in [25]. The testing datasets include *BMS-WEBVIEW-1*, *mushroom*, *pumsb_star* and *T10I4D100K*. These datasets form a good representative of both synthetic and real datasets. The detailed characteristics of the datasets are presented in Table 3.1.

The performance of previously proposed maintenance algorithms is tested with incremental updates. This is because most of previous algorithms can only handle incremental updates but not decremental updates and support threshold adjustment. The processing time of previous algorithms is measured under the setting of batch maintenance. The efficiency of previous algorithms is tested over various update intervals. The incremental update interval, denoted as Δ^+, refers to the ratio between the sizes of the incremental dataset and original dataset. $\Delta^+ = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

Dataset	Size	#Trans	#Items	maxTL	aveTL
BMS-WEBVIEW-1	0.99MB	59,602	497	268	2.51
mushroom	0.56MB	8,124	119	23	23
pumsb_star	11.03MB	49,046	2,088	63	50.48
T10I4D100K	3.93MB	100,000	870	30	10.10

Table 3.1: Characteristics of datasets: *BMS-WEBVIEW-1*, *mushroom*, *pumsb_star*, *T10I4D100K*. Notations: #Trans denotes the total number of transactions in the dataset, #Items denotes the total number of distinct items, maxTL denotes the maximal transaction length and aveTL is the average transaction length.

The effectiveness of previous algorithms is compared with two representative frequent pattern discovery algorithms — Apriori and FP-growth. In this thesis, the implementation of Apriori by Bart Goethals [24] is used, and FPgrowth* [28], one of the fastest implementation of FP-growth [25], is used. Given an incremental dataset $\mathcal{D}_{inc}$, we first obtain the updated dataset $\mathcal{D}_{upd+}$ by merging $\mathcal{D}_{org}$ and $\mathcal{D}_{inc}$. Apriori and FPgrowth* are then applied to re-discover the frequent pattern space of $\mathcal{D}_{upd+}$. In this study, we assume the time for dataset merging is trivial compared to the discovery time and thus can be ignored.

Figure 3.4 illustrates the performance of Apriori-based maintenance algorithms, FUP2H and Borders, and partition-based maintenance algorithm, SWF. It is observed that maintenance algorithms outperform discovery algorithm Apriori over various update intervals. In particular, FUP2H is on average twice faster than Apriori, and, especially for dataset *mushroom*, FUP2H is up to 5 times faster than Apriori . Borders is much more effective compared with FUP2H. Borders, on average, outperforms Apriori by an order of magnitude. This experimentally justified that the "border pattern" is a useful concept that helps to avoid redundant enumeration of existing frequent patterns. The performance of SWF falls in between FUP2H and Borders.

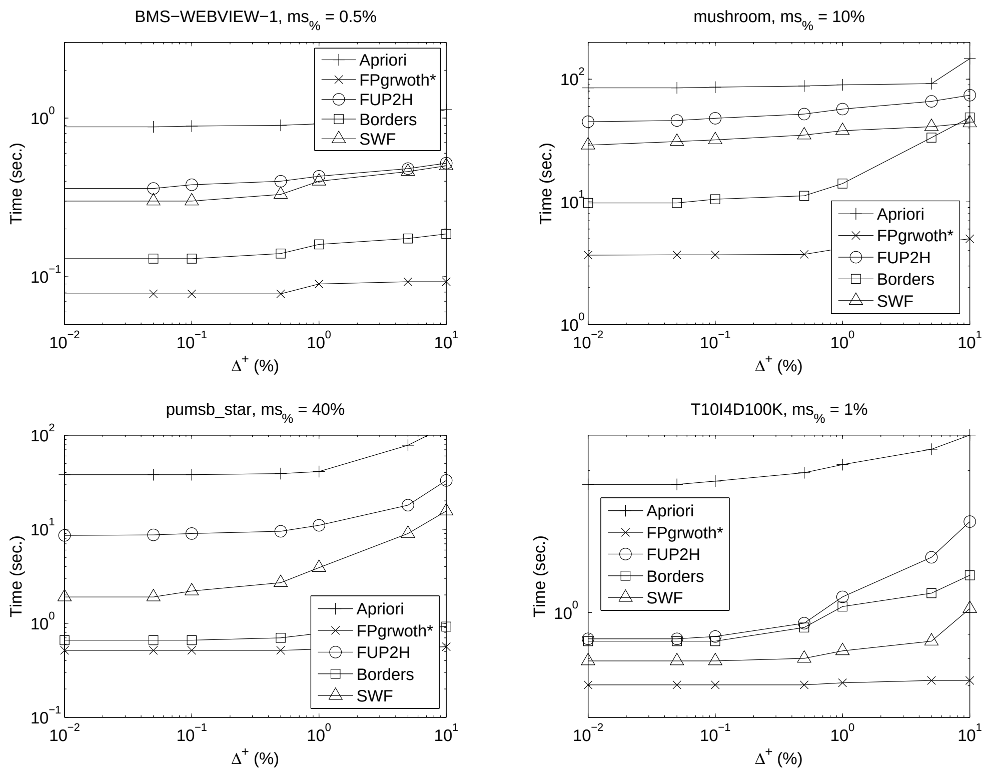


Figure 3.4: Performance of FUP2H, Borders and SWF over various update intervals. Notation: $\Delta^{+} = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

SWF is around 6 times faster than Apriori in general. However, FUP2H, Borders and SWF are much slower compared with FPgrowth*, the prefix-tree based discovery algorithm. FUP2H, Borders and SWF all employs the candidate-generation-verification framework. Although equipped with certain candidate pruning techniques, they all inevitably generate unnecessary candidates. On the other hand, as illustrated in Chapter 2, FPgrowth* effectively discovers the frequent pattern space without generation candidate patterns.

Figure 3.5 summarizes the performance of prefix-tree based maintenance algorithm, CanTree, and concise-representation based maintenance algorithms, ZIGZAG and moment. It can be seen in Figure 3.5 that CanTree, ZIGZAG and moment are more effective maintenance algorithms compared with FUP2H, Borders and SWF.

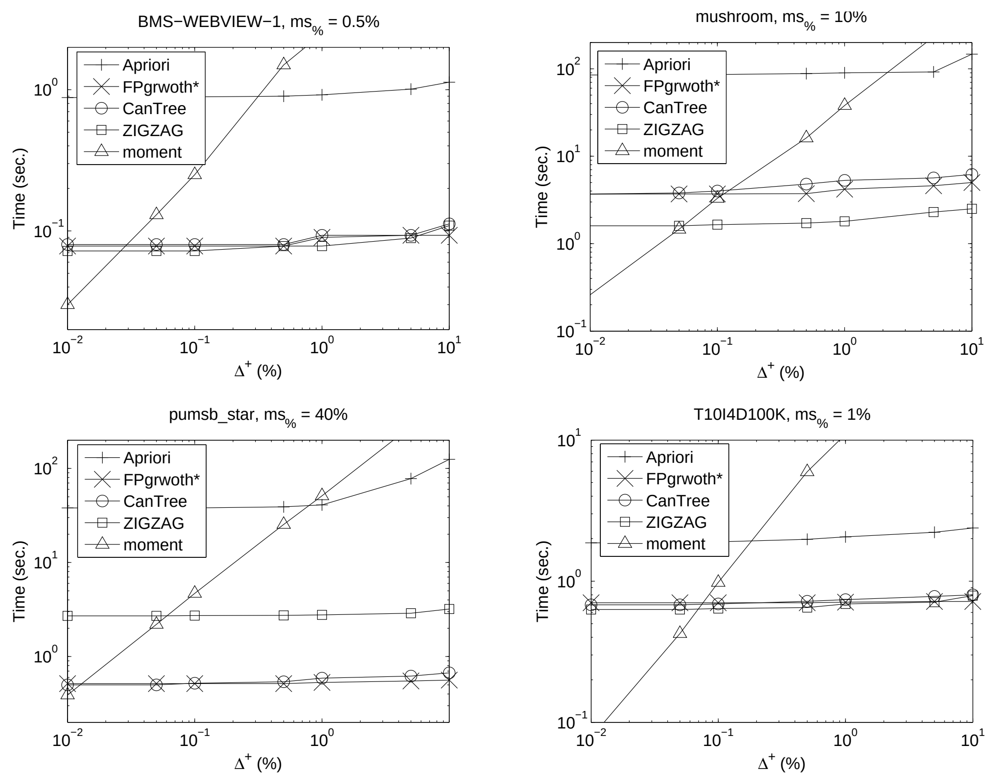


Figure 3.5: Performance of CanTree, ZIGZAG and moment over various update intervals. Notation: $\Delta^+ = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

CanTree, on average, outperforms Apriori by an order of magnitude. CanTree performs the best for dataset *mushroom*, where it is almost one thousand times faster than Apriori. Moreover, we observe that the performance of CanTree is very closed to but slightly slower the one of FPgrowth* (an implementation of FP-growth). This observation is contrary to the one in [44]. In [44], it is shown that CanTree outperforms FP-growth, and the advantage of CanTree gets larger as the update interval increases. The differences between our observation and the observation in [44] may mainly due to two reasons. First, we may use a different implementation of FP-growth from [44]. Second, we employ a different re-discovery approach. In [44], two *FP-tree*s, one for the original dataset $\mathcal{D}_{org}$ and the other for the updated dataset $\mathcal{D}_{upd+}$, need to be constructed to re-discover the frequent pattern space. However, in our re-discovery approach, only one *FP-tree* for the updated dataset $\mathcal{D}_{upd+}$ is required. We believe that our re-discovery approach is more straightforward and computationally effective.

ZIGZAG maintains only the maximal frequent patterns, where the number of involved patterns is much smaller than the size of frequent pattern space. Therefore, ZIGZAG is much more effective compared with Apriori. ZIGZAG outperforms Aprirori on average more than an order of magnitude. However, comparing with FPgrowth*, the speed up gained by ZIGZAG varies from dataset to dataset. For example, ZIGZAG is approximately 3 times faster FPgrowth* for dataset *mushroom*, but ZIGZAG is twice slower for dataset *pumsb_star*.

Algorithm moment maintains the frequent closed patterns. The set of frequent closed patterns, similar to frequent maximal patterns, is much smaller than the entire space of frequent patterns. As a result, as graphically demonstrated in Figure 3.5, moment is much faster than the discovery algorithms when the update interval is small ($\Delta^{+} < 0.1\%$). Take dataset *mushroom* as an example: moment outperforms Apriori up to 3 orders of magnitude, and it is more than an order of magnitude faster than FPgrowth* for the best cases. Nevertheless, we also observe that the performance of moment degrades dramatically as the update interval increases. This observation is consistent with our algorithmic analysis in the previous section. As the update interval gets large, the hypothesis of moment becomes no longer valid, and thus the efficiency of moment drops significantly.

3.3 Summary & Discussion

This chapter reviewed the representative maintenance algorithms for frequent patterns. The strengths and weaknesses of these algorithms are summarized and compared in Table 3.2-3.4.

We observe that most existing algorithms are proposed as an extension of certain frequent pattern discovery algorithms or the data structure they used. No prior work has investigated in details how the frequent pattern space evolves in response to updates. We believe that understanding the space evolution is necessary and important for the development of effective maintenance algorithms. As a result, the evolution of frequent pattern space under various update scenarios is theoretically studied in the following chapter.

	Algorithm	Strengths	Weakness
Apriori-based	FUP & FUP2H	Make use of the support information of the previously discovered frequent patterns to reduce the number of candidate patterns.	(1) Generate a large amount of unnecessary candidates. (2) Require multiple scans of datasets.
	Borders	(1) Avoids enumeration of previous discovered patterns. (2) Enumerates new frequent patterns efficiently from the border patterns.	(1) Generates a large amount of unnecessary candidates. (2) Requires multiple scans of datasets.
	DELI	Estimates the impact of data updates on the frequent pattern space via sampling and statistical methods. Avoids unnecessary updates, if the impact is negligible.	Generates only an approximate set of frequent patterns with estimated support values (not the focus of this Thesis).
Partition-based	SWF	(1) Prunes most of the false candidates in the early stage based on the "partitioning heuristic". (2) Requires only one full scan of dataset.	Generates unnecessary candidates.

Table 3.2: Summary of *Apriori-based* and *Partition-based* maintenance algorithms.

	Algorithm	Strengths	Weakness
Prefix-tree-based	AFPIM	(1) Summarizes dataset into a prefix-tree and requires only two scans of the dataset. (2) Enumerates no false candidates.	(1) Requires re-organization of the entire tree for each update. (2) Has to rebuild the prefix-tree if new frequent items emerge.
	CATS Tree	(1) Point (1) & (2) of AFPIM. (2) Sorts items locally, which allows the tree to be locally updated. (3) Allows newly emerged items to be included easily.	Requires node swapping and merging, which are computational expensive, for the local update of prefix-tree.
	CanTree	(1) Point (1) & (2) of AFPIM. (2) Arranges items in a canonical-order, which will not be affected by the data update, so that no re-sorting, node swapping and node merging are needed.	Less compact compared to CATS tree.

Table 3.3: Summary of *Prefix-tree-based* maintenance algorithms.

	Algorithm	Strengths	Weakness
Concise-representation-based	ZIGZAG	(1) Updates the maximal frequent patterns with a backtracking search. (2) Prunes infrequent and non-maximal patterns in the early stage.	Maximal patterns are lossy representations of frequent patterns.
	moment	(1) Maintains only the frequent closed patterns. (2) Employs a novel data structure, *CET*, to effectively store and update the frequent closed patterns and boundary patterns. (3) Prunes non-closed candidate patterns in the early stage.	Restricted by the hypothesis that there is only a small amount of data updates and thus small changes to frequent closed patterns.

Table 3.4: Summary of *Concise-representation-based* maintenance algorithms.

Part II

THEORIES

Chapter 4

Frequent Pattern Space Evolution: A Theoretical Analysis

This chapter analyzes how the space of frequent patterns evolves in response to various updates. However, due to the vast size of the frequent pattern space, direct analysis on the pattern space is extremely difficult. To solve this problem, we propose to structurally decompose the frequent pattern space into sub-spaces — equivalence classes. We then study the space evolution based on the concept of equivalence classes. We investigate the space evolution under three update scenarios: incremental updates, decremental updates and support threshold adjustment.

4.1 Structural Decomposition of Pattern Space

The space of frequent patterns can be large. However, this space possesses the nice convexity property, which is very helpful when it comes to concise and lossless representation of the space and its subspaces.

Definition 4.1 (Convexity). *A space $\mathcal{S}$ is convex if, for all $X, Y \in \mathcal{S}$ such that $X \subseteq Y$, it is the case that $Z \in S$ whenever $X \subseteq Z \subseteq Y$.*

For a convex space $\mathcal{S}$, we define the collection of all "most general" patterns in $\mathcal{S}$ as the "left bound" of $\mathcal{S}$, denoted $\mathcal{L}$. A pattern X is most general in $\mathcal{S}$ if there is no proper subset of X in $\mathcal{S}$. Similarly, we define the collection of all "most specific" patterns as the "right bound" of $\mathcal{S}$, denoted $\mathcal{R}$. A pattern X is most specific in $\mathcal{S}$ if there is no proper superset of X in $\mathcal{S}$. We call the pair of left and right bound the "border" of $\mathcal{S}$, which is denoted by $\langle \mathcal{L}, \mathcal{R} \rangle$. We also define $[\mathcal{L}, \mathcal{R}] = \{Z \mid$ there is $X \in \mathcal{L}$, there is $Y \in \mathcal{R}$, $X \subseteq Z \subseteq Y\}$. $\langle \mathcal{L}, \mathcal{R} \rangle$ and $[\mathcal{L}, \mathcal{R}]$ are two different notions. Specifically, $[\mathcal{L}, \mathcal{R}] = \mathcal{S}$, but $\langle \mathcal{L}, \mathcal{R} \rangle$ is only a concise representation of the whole space $\mathcal{S}$ in a lossless way.

Fact 4.2 (Cf. [21]). *$\mathcal{F}(\mathcal{D}, ms_{\%})$ is convex. Furthermore, its border is of the form $\langle \{\emptyset\}, \mathcal{R} \rangle$ for some $\mathcal{R}$.*

Algorithms Max-Miner [35] and GENMAX [27] were proposed to discover frequent maximal patterns — the $\mathcal{R}$ in Fact 4.2.

We further discover that convex frequent pattern spaces can be further decomposed systematically into convex sub-spaces based on the concept of equivalence classes.

Definition 4.3 (Equivalence class)**.** *Let the "filter", $f(P, \mathcal{D})$, of a pattern P in a dataset $\mathcal{D}$ be defined as $f(P, \mathcal{D}) = \{T \in \mathcal{D} \mid P \subseteq T\}$. Then the "equivalence class" $[P]_{\mathcal{D}}$ of P in a dataset $\mathcal{D}$ is the collection of patterns defined as $[P]_{\mathcal{D}} = \{Q \mid f(P, \mathcal{D}) = f(Q, \mathcal{D})\}$.*

Fact 4.4 (Cf. [47])**.** *Given a dataset $\mathcal{D}$ and minimum support threshold $ms_{\%}$, the frequent pattern space $\mathcal{F}(ms_{\%}, \mathcal{D}) = \bigcup_{i=1}^{n}[P_i]_{\mathcal{D}}$, where $[P_i] \cap [P_j] = \emptyset$ for $1 \leq i < j \leq n$.*

According to Definition 4.3, two patterns are "equivalent" in the context of a dataset $\mathcal{D}$ iff they are included in exactly the same transactions in $\mathcal{D}$. Thus the patterns in the same equivalence class have the same support. So we extend our notations and write $sup(P, \mathcal{D})$ to denote the support of equivalence class $[P]_{\mathcal{D}}$ and $P \in \mathcal{F}(\mathcal{D}, ms_{\%})$ to mean the equivalence class is frequent. A nice property of equivalence classes of patterns is that they are convex.

Fact 4.5 (Cf. [47])**.** *Equivalence class $[P]_{\mathcal{D}}$ is convex, and the right bound of its border is singleton set.*

Together with equivalence classes, frequent "closed patterns" and frequent "generators" (also called "key patterns") have been widely studied in the data mining field. We discuss next the relationship between equivalence classes and closed patterns and generators.

As discussed, every equivalence class is convex, and its right bound is the most specific pattern in the class. Thus all supersets of its right bound necessarily have lesser support. Therefore, the most specific pattern in an equivalence class is a closed pattern (Definition 2.3). Conversely, all supersets of a closed pattern necessarily have lesser support. Therefore a closed pattern is the right bound of its equivalence class. Similarly, the left bound of an equivalence class are all the most general patterns in the class. Thus every subset of each of these most general patterns necessarily has higher support. Therefore, the most general patterns in an equivalence class are all generators (Definition 2.3). Conversely, the subsets of a generator have higher support by definition. So these subsets are not in the same equivalence class as the generator. Hence, a generator is one of the left bound patterns of its equivalence class. So we have the following alternative equivalent definitions for generators and closed patterns.

Fact 4.6. *A pattern P is a generator or key pattern in a dataset $\mathcal{D}$ iff P is a most general pattern in $[P]_{\mathcal{D}}$. A pattern P is a closed pattern in a dataset $\mathcal{D}$ iff P is the most specific pattern in $[P]_{\mathcal{D}}$.*

Every subset of a frequent pattern is also a frequent pattern. This is the well-known "*a priori*" property. Generators of equivalence classes also enjoy the "*a priori*" property.

Fact 4.7. *Let P be a pattern in $\mathcal{D}$. If P is frequent, then every subset of P is also frequent. If P is a generator (or key pattern), then every subset of P is also a generator (or key pattern) in $\mathcal{D}$. Thus, if P is a frequent generator (or key pattern), then every subset of P is also a frequent generator (or key pattern) in $\mathcal{D}$.*

Proof. Suppose P is frequent. Since every subset of P must occur in every transaction that P occurs in, it is clear that every subset of P has a support at least that of P. So every subset of P is also frequent.

Suppose P is a generator in $\mathcal{D}$. Suppose $Q \subseteq P$. We want to show Q is also a generator. Suppose there is a $R \subseteq Q$ in $[Q]_{\mathcal{D}}$. So there is an S such that $R = Q - S$, and $S \subseteq Q$. Let $T = P - S$. Let D be a transaction having T. We have $R \subseteq T$, as $Q \subseteq P$ and thus $Q - S \subseteq P - S$. Then D has R as well, as $R \subseteq T$. Then D has Q as well, as $R \in [Q]_{\mathcal{D}}$. Then D has S also, as $S \subseteq Q$. Then D has P. This means every transaction having T also has P. We already know that every transaction having P also has T. So, P and T are in the same equivalence class. Since P is a generator, and $T = P - S$, it must be the case that $S = \{\}$. Since $R = Q - S$, we conclude $R = Q$. Hence Q is a generator. $\square$

On the other hand, closed patterns also enjoy certain "anti-monotonicity" property, in the sense that a closed pattern is a subset of another closed pattern whenever the filter of the latter is a subset of the former. Similarly, equivalence classes of patterns enjoy the "anti-monotonicity" property.

Fact 4.8. 1. *Let P and Q be closed patterns in $\mathcal{D}$. Then $P \subseteq Q$ iff $f(Q, \mathcal{D}) \subseteq f(P, \mathcal{D})$.*

2. *Suppose $\mathcal{D}' \subseteq \mathcal{D}$. Then $[P]_{\mathcal{D}} \subseteq [P]_{\mathcal{D}'}$ iff $f(P, \mathcal{D}') \subseteq f(P, \mathcal{D})$.*

The most useful feature of closed patterns and generators is that they form the borders of corresponding equivalence classes. Therefore, they uniquely define the corresponding equivalence class. This implies that, to mine or maintain generators and closed patterns, it is sufficient to mine or maintain the borders of equivalence classes, and vice versa. Figure 4.1 graphically demonstrates how the pattern space can be structurally decomposed into equivalence classes and how an equivalence class

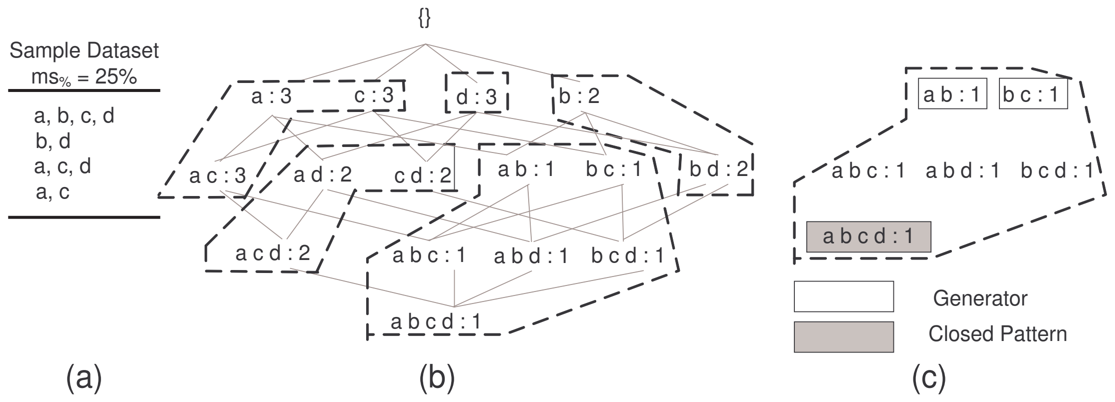


Figure 4.1: Demonstration of the structural decomposition of the frequent pattern space. (a)The sample dataset; (b) decomposition of the frequent pattern space of the sample dataset into 5 equivalence classes; and (c) the "border" of an equivalence class.

can be concisely represented by its border.

The equivalence class is an effective concise representation for pattern spaces. In the literature, the equivalence class has been used to summarize cells in data cubes [45, 46]. Here we use equivalence classes to concisely represent the space of frequent patterns. Structurally decomposing the pattern space into equivalence classes allows us to investigate the evolution of the pattern space via studying the evolution of equivalence classes, which is much smaller and easier to study. Moreover, the structural decomposition simplifies the maintenance problem from updating the entire space to the update of equivalence classes, and it also allows us to maintain the pattern space in a divide-and-conquer manner.

4.2 Space Evolution under Incremental Updates

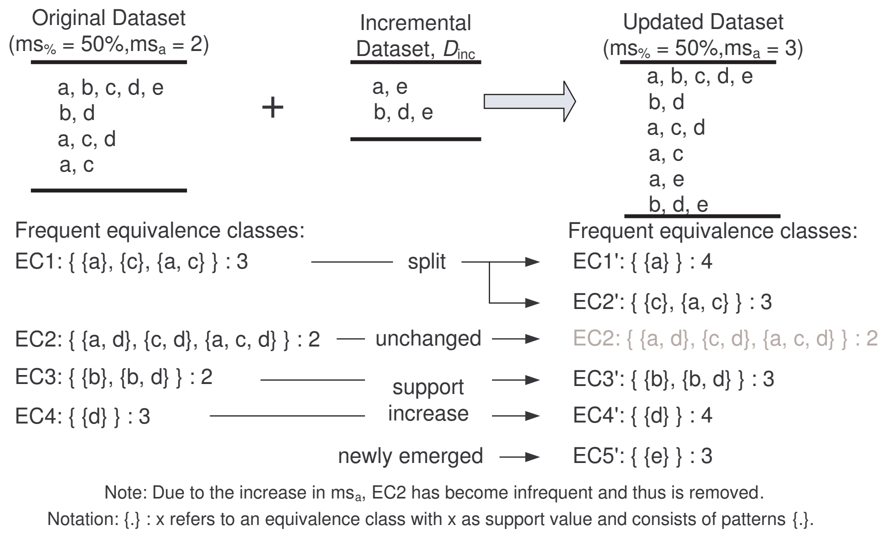
We investigate in this section how the space of frequent patterns, the equivalence classes, and their support values evolve when a dataset is incrementally updated with multiple new transactions. For this section, we use the following notations: $\mathcal{D}_{org}$ is the original dataset, $\mathcal{D}_{inc}$ is the set of new transactions to be added to $\mathcal{D}_{org}$, and $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \mathcal{D}_{inc}$ is the updated dataset. We assume without loss of generality that $\mathcal{D}_{org} \neq \emptyset$ and $\mathcal{D}_{org} \cap \mathcal{D}_{inc} = \emptyset$. This leads to the conclusion that, for any pattern P, $f(P, \mathcal{D}_{upd+}) = f(P, \mathcal{D}_{org}) \cup f(P, \mathcal{D}_{inc})$ and $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{inc})$.

We first show that a pattern that is a generator, or is a closed pattern, remains to be a generator, or closed pattern respectively when multiple transactions are added to the underlying dataset.

Proposition 4.9. *Let P be a pattern in $\mathcal{D}_{org}$. If P is a generator in $\mathcal{D}_{org}$, then P is a generator in $\mathcal{D}_{upd+}$. If P is a closed pattern in $\mathcal{D}_{org}$, then P is a closed pattern in $\mathcal{D}_{upd+}$.*

Proof. Suppose P is a generator in $\mathcal{D}_{org}$. Let Q be any pattern that $Q \subset P$. Then $sup(Q, \mathcal{D}_{org}) > sup(P, \mathcal{D}_{org})$, and $sup(Q, \mathcal{D}_{inc}) \geq sup(P, \mathcal{D}_{inc})$. Hence $sup(Q, \mathcal{D}_{upd+}) = sup(Q, \mathcal{D}_{org}) + sup(Q, \mathcal{D}_{inc}) > sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{inc}) = sup(P, \mathcal{D}_{upd+})$. Thus, P is a generator in $\mathcal{D}_{upd+}$ as desired.

Next, suppose P is a closed pattern in $\mathcal{D}_{org}$. Let Q be any pattern that $Q \supset P$. Then $sup(Q, \mathcal{D}_{org}) < sup(P, \mathcal{D}_{org})$, and $sup(Q, \mathcal{D}_{inc}) \leq sup(P, \mathcal{D}_{inc})$. Hence $sup(Q, \mathcal{D}_{upd+}) = sup(Q, \mathcal{D}_{org}) + sup(Q, \mathcal{D}_{inc}) < sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{inc}) = sup(P, \mathcal{D}_{upd+})$. Thus, P is a closed pattern in $\mathcal{D}_{upd+}$ as desired. $\square$



Original Dataset
(ms% = 50%,msa = 2)
a, b, c, d, e
b, d
a, c, d
a, c
+
Incremental Dataset, Dinc
a, e
b, d, e
Updated Dataset
(ms% = 50%,msa = 3)
a, b, c, d, e
b, d
a, c, d
a, c
a, e
b, d, e
Frequent equivalence classes:
EC1: { {a}, {c}, {a, c} } : 3
split
EC1': { {a} } : 4
EC2': { {c}, {a, c} } : 3
EC2: { {a, d}, {c, d}, {a, c, d} } : 2
unchanged
EC2: { {a, d}, {c, d}, {a, c, d} } : 2
EC3: { {b}, {b, d} } : 2
support increase
EC3': { {b}, {b, d} } : 3
EC4: { {d} } : 3
EC4': { {d} } : 4
newly emerged
EC5': { {e} } : 3
Frequent equivalence classes:
Note: Due to the increase in msa, EC2 has become infrequent and thus is removed.
Notation: {.} : x refers to an equivalence class with x as support value and consists of patterns {.}.

Figure 4.2: Evolution of equivalence classes under incremental updates.

Generators and closed patterns are preserved by incremental updates. This implies that some equivalence classes, as identified by their unique closed patterns, are also preserved by incremental updates. We observe that an existing equivalence class can evolve in exactly three ways under incremental updates, as shown in Figure 4.2. The first way is to remain unchanged without any change in support. The second way is to remain unchanged but with an increased support. The third way is to shrink—by splitting into two or more classes, where at most one of the resulting classes has the same closed pattern and same support as the original equivalence class and all other resulting classes have higher support. In short, after an incremental update, the support of an equivalence class can only increase and the size of an equivalence class can only shrink by splitting.

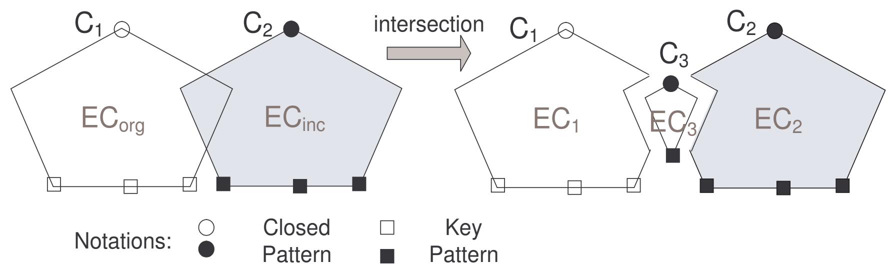


Figure 4.3: Splitting of equivalence classes.

Proposition 4.10. *Let P be a pattern in $\mathcal{D}_{org}$. Then $[P]_{\mathcal{D}_{upd+}} \subseteq [P]_{\mathcal{D}_{org}}$ and $sup(P, \mathcal{D}_{upd+}) \geq sup(P, \mathcal{D}_{org})$.*

Proof. Suppose $Q \in [P]_{\mathcal{D}_{upd+}}$. Then $f(Q, \mathcal{D}_{upd+}) = f(Q, \mathcal{D}_{org}) \cup f(Q, \mathcal{D}_{inc}) = f(P, \mathcal{D}_{upd+}) = f(P, \mathcal{D}_{org}) \cup f(P, \mathcal{D}_{inc})$. Since $\mathcal{D}_{inc} \cup \mathcal{D}_{org} = \emptyset$, we have $f(Q, \mathcal{D}_{org}) = f(P, \mathcal{D}_{org})$. Then $Q \in [P]_{\mathcal{D}_{org}}$ for every $Q \in [P]_{\mathcal{D}_{upd+}}$. Thus we can conclude $[P]_{\mathcal{D}_{upd+}} \subseteq [P]_{\mathcal{D}_{org}}$. Also, $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{inc}) \geq sup(P, \mathcal{D}_{org})$. □

Let us now illustrate the splitting of an equivalence class in more detail. As shown in Figure 4.3, an equivalence class EC_{org} in the original dataset overlaps with an equivalence class EC_{inc} in the incremental dataset. Therefore, after the update, EC_{org} is split into two parts, denoted as EC_1 and EC_3. EC_3 is an equivalence class in $\mathcal{D}_{upd+}$, which has a newly emerged closed pattern C_3 and a newly emerged generator. The messy parts are EC_1 and EC_2. This is because EC_1 and EC_2 are not convex spaces, as shown in Figure 4.3. However, as we will see in Theorem 4.11 and Corollary 4.15, EC_1 and EC_2 will further split into multiple convex equivalence classes.

Theorem 4.11. *Let P be a pattern in $\mathcal{D}_{org}$. Then $[P]_{\mathcal{D}_{org}}$ splits into n classes $[Q_1]_{\mathcal{D}_{upd+}}$, ..., $[Q_n]_{\mathcal{D}_{upd+}}$ iff $f(Q_i, \mathcal{D}_{inc}) \neq f(Q_j, \mathcal{D}_{inc})$ for $1 \leq i < j \leq n$, where $Q_i \in [P]_{\mathcal{D}_{org}}$ for $1 \leq i \leq n$, and n is the largest such integer. Furthermore, $[Q_i]_{\mathcal{D}_{upd+}} = [Q_i]_{\mathcal{D}_{inc}} \cap [P]_{\mathcal{D}_{org}}$ for $1 \leq i \leq n$. Moreover, there is at most one Q_i among Q_1, ..., Q_n, such that Q_i does not occur in $\mathcal{D}_{inc}$ and $Q_i \in [C]_{\mathcal{D}_{upd+}}$, where C is the closed pattern of $[P]_{\mathcal{D}_{org}}$.*

Proof. We first prove $[Q_i]_{\mathcal{D}_{upd+}} = [Q_i]_{\mathcal{D}_{inc}} \cap [P]_{\mathcal{D}_{org}}$ for $1 \leq i \leq n$. Recall that $[Q_i]_{\mathcal{D}_{upd+}} = \{Q | f(Q, \mathcal{D}_{upd+}) = f(Q_i, \mathcal{D}_{upd+})\} = \{Q | f(Q, \mathcal{D}_{org}) \cup f(Q, \mathcal{D}_{inc}) = f(Q_i, \mathcal{D}_{org}) \cup f(Q_i, \mathcal{D}_{inc})\}$ for $1 \leq i \leq n$. Since $\mathcal{D}_{org} \cap \mathcal{D}_{inc} = \emptyset$, we can alternatively define that $[Q_i]_{\mathcal{D}_{upd+}} = \{Q | f(Q, \mathcal{D}_{org}) = f(Q_i, \mathcal{D}_{org}) \wedge f(Q, \mathcal{D}_{inc}) = f(Q_i, \mathcal{D}_{inc})\} = \{Q | Q \in [Q_i]_{\mathcal{D}_{org}} \wedge Q \in [Q_i]_{\mathcal{D}_{inc}}\}$ for $1 \leq i \leq n$. Therefore, $[Q_i]_{\mathcal{D}_{upd+}} = [Q_i]_{\mathcal{D}_{inc}} \cap [P]_{\mathcal{D}_{org}}$, for $1 \leq i \leq n$, as desired.

Second, we prove the left-to-right direction of the theorem. Suppose $[P]_{\mathcal{D}_{org}}$ splits into $[Q_1]_{\mathcal{D}_{upd+}}$, ..., $[Q_n]_{\mathcal{D}_{upd+}}$. By the usual definition of splits, we have $[P]_{\mathcal{D}_{org}} = \bigcup_{1 \leq i \leq n} [Q_i]_{\mathcal{D}_{upd+}}$, $[Q_i]_{\mathcal{D}_{upd+}} \cap [Q_j]_{\mathcal{D}_{upd+}} = \emptyset$ for $1 \leq i < j \leq n$, and $Q_i \in [P]_{\mathcal{D}_{org}}$ for $1 \leq i \leq n$. Since $[Q_i]_{\mathcal{D}_{upd+}} \cap [Q_j]_{\mathcal{D}_{upd+}} = \emptyset$ for $1 \leq i < j \leq n$, then $f(Q_i, \mathcal{D}_{upd+}) = f(Q_i, \mathcal{D}_{org}) \cup f(Q_i, \mathcal{D}_{inc}) \neq f(Q_j, \mathcal{D}_{upd+}) = f(Q_j, \mathcal{D}_{org}) \cup f(Q_j, \mathcal{D}_{inc})$ for $1 \leq i < j \leq n$. Also since $Q_i \in [P]_{\mathcal{D}_{org}}$ for $1 \leq i \leq n$, we have $f(Q_i, \mathcal{D}_{org}) = f(Q_j, \mathcal{D}_{org})$ for $1 \leq i < j \leq n$. Thus it must be true that $f(Q_i, \mathcal{D}_{inc}) \neq f(Q_j, \mathcal{D}_{inc})$ for $1 \leq i < j \leq n$. This n is also the largest integer having this property. Otherwise, there is a $Q \notin \bigcup_{1 \leq i \leq n} [Q_i]_{Dupd+}$ such that $Q \in [P]_{Dorg}$, contradicting the definition of splits. This proves the left-to-right direction.

Third, we prove the right-to-left direction of the theorem. Suppose $f(Q_i, \mathcal{D}_{inc}) \neq f(Q_j, \mathcal{D}_{inc})$ for $1 \leq i < j \leq n$, where $Q_i \in [P]_{\mathcal{D}_{org}}$ for $1 \leq i \leq n$. This implies that $[Q_i]_{\mathcal{D}_{inc}} \cap [Q_j]_{\mathcal{D}_{inc}} = \emptyset$ for $1 \leq i < j \leq n$. Then based on the definition of Q_i and Q_j for $1 \leq i < j \leq n$ (n is the largest such integer), it is clear that $[P]_{\mathcal{D}_{org}} \subseteq \bigcup_{1 \leq i \leq n} [Q_i]_{\mathcal{D}_{inc}}$. Thus we can formulate $[P]_{\mathcal{D}_{org}} = \bigcup_{1 \leq i \leq n} [Q_i]_{\mathcal{D}_{inc}} \cap [P]_{\mathcal{D}_{org}} = \bigcup_{1 \leq i \leq n} ([Q_i]_{\mathcal{D}_{inc}} \cap [P]_{\mathcal{D}_{org}}) = \bigcup_{1 \leq i \leq n} [Q_i]_{\mathcal{D}_{upd+}}$ as desired.

It remains to show that there is at most one Q_i among Q_1, ..., Q_n, such that Q_i does not occur in $\mathcal{D}_{inc}$. Let C be the closed pattern of $[P]_{\mathcal{D}_{org}}$. It is clear that if C occurs in $\mathcal{D}_{inc}$, then all of Q_1, ..., Q_n occur in $\mathcal{D}_{inc}$. So we can assume without loss of generality that C does not occur in $\mathcal{D}_{inc}$ and that $C \in [Q_1]_{\mathcal{D}_{upd+}}$. Thus $[Q_1]_{\mathcal{D}_{upd+}} = \{Q \in [P]_{\mathcal{D}_{org}} | f(Q, \mathcal{D}_{inc}) = \emptyset\}$. Then we assume there is one Q_j among $Q_2, \cdots, Q_n$ such that Q_j does not occur in $\mathcal{D}_{inc}$ neither. So we have $[Q_j]_{\mathcal{D}_{upd+}} = \{Q \in [P]_{\mathcal{D}_{org}} | f(Q, \mathcal{D}_{inc}) = \emptyset\} = [Q_1]_{\mathcal{D}_{upd+}}$. This contradicts to the condition that $[Q_i]_{\mathcal{D}_{upd+}} \cap [Q_j]_{\mathcal{D}_{upd+}} = \emptyset$. Thus we can conclude that Q_1 is the unique one that does not occur in $\mathcal{D}_{inc}$. It is also clear that $Q_1 \in [C]_{\mathcal{D}_{upd+}}$. $\square$

It follows from Theorem 4.11 that $[P]_{\mathcal{D}_{org}}$ is split into a number of equivalence classes in $\mathcal{D}_{upd+}$, one for each group of patterns in $[P]_{\mathcal{D}_{org}}$ that appear in the same transactions in $\mathcal{D}_{inc}$. Each resulting new equivalence class $[Q_i]_{\mathcal{D}_{upd+}}$ has a higher support if Q_i occurs in $\mathcal{D}_{inc}$, and has the same support if Q_i does not occur in $\mathcal{D}_{inc}$. Consequently, if all patterns in $[P]_{\mathcal{D}_{org}}$ occur in exactly the same transactions in $\mathcal{D}_{inc}$, then $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$ with higher support. On the other hand, if no pattern in $[P]_{\mathcal{D}_{org}}$ occurs in any transaction in $\mathcal{D}_{inc}$, then $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$ with no change in support.

Corollary 4.12. *Let P be a pattern in $\mathcal{D}_{org}$. Then $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$ and $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org})$ iff $f(Q, \mathcal{D}_{inc}) = \emptyset$ for every pattern $Q \in [P]_{\mathcal{D}_{org}}$.*

Corollary 4.13. *Let P be a pattern in $\mathcal{D}_{org}$. Then $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$ and $sup(P, \mathcal{D}_{upd+}) > sup(P, \mathcal{D}_{org})$ iff $f(Q, \mathcal{D}_{inc}) = f(P, \mathcal{D}_{inc}) \neq \emptyset$ for every $Q \in [P]_{\mathcal{D}_{org}}$.*

Incremental updates also induce new equivalence classes to emerge. It is worth noting a symmetry between $\mathcal{D}_{org}$ and $\mathcal{D}_{inc}$. Specifically, it is equally reasonable to think of $\mathcal{D}_{upd+}$ as being obtained by adding $\mathcal{D}_{org}$ to $\mathcal{D}_{inc}$, exchanging the roles of $\mathcal{D}_{org}$ and $\mathcal{D}_{inc}$. Thus we can describe the emergence of new equivalence classes with the following corollaries, which are indeed the counterparts of Corollary 4.12 and Theorem 4.11.

Corollary 4.14. *Let P be a pattern in $\mathcal{D}_{inc}$. Then $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{inc}}$ and $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{inc})$ iff $f(Q, \mathcal{D}_{org}) = \emptyset$ for every pattern $Q \in [P]_{\mathcal{D}_{inc}}$.*

Corollary 4.15. *Let P be a pattern in $\mathcal{D}_{inc}$. Then $[P]_{\mathcal{D}_{inc}}$ splits into n classes $[Q_1]_{\mathcal{D}_{upd+}}$, ..., $[Q_n]_{\mathcal{D}_{upd+}}$ iff $f(Q_i, \mathcal{D}_{org}) \neq f(Q_j, \mathcal{D}_{org})$ for $1 \leq i < j \leq n$, where $Q_i \in [P]_{\mathcal{D}_{inc}}$ for $1 \leq i \leq n$, and n is the largest such integer. Furthermore, $[Q_i]_{\mathcal{D}_{upd+}} = [Q_i]_{\mathcal{D}_{org}} \cap [P]_{\mathcal{D}_{inc}}$ for $1 \leq i \leq n$. Moreover, there is at most one Q_i among Q_1, ..., Q_n, such that Q_i does not occur in $\mathcal{D}_{org}$, meaning $[Q_i]_{\mathcal{D}_{upd+}}$ consists of entirely new patterns, and furthermore $Q_i \in [C]_{\mathcal{D}_{upd+}}$, where C is the closed pattern of $[P]_{\mathcal{D}_{inc}}$.*

Combining Corollaries 4.15, 4.14, 4.13, 4.12, and Theorem 4.11, we can characterize the evolution of the frequent pattern space under incremental updates based on equivalence classes in $\mathcal{D}_{inc}$ as follows. We use $Close(X)$ and $Keys(X)$ to denote the closed pattern and generators (key patterns) of the equivalence class X.

Theorem 4.16. *For every equivalence class $[P]_{\mathcal{D}_{upd+}}$ in $\mathcal{D}_{upd+}$, exactly one of the 6 scenarios below holds:*

1. *P is in $\mathcal{D}_{inc}$, P is not in $\mathcal{D}_{org}$, and Q is not in $\mathcal{D}_{org}$ for all $Q \in [P]_{\mathcal{D}_{inc}}$, corresponding to the scenario where an equivalence class comprising entirely of new patterns has emerged. In this case, $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{inc}}$, $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{inc})$, $Close([P]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{inc}})$ and $Keys([P]_{\mathcal{D}_{upd+}}) = Keys([P]_{\mathcal{D}_{inc}})$.*

2. *P is in $\mathcal{D}_{inc}$, P is not in $\mathcal{D}_{org}$, and Q is in $\mathcal{D}_{org}$ for some $Q \in [P]_{\mathcal{D}_{inc}}$, corresponding to the scenario where $[P]_{\mathcal{D}_{inc}}$ has to be split, and an equivalence class comprising entirely of new patterns has emerged, as stated in Corollar 4.15. Then, $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{inc}} - \bigcup\{[Q]_{\mathcal{D}_{org}} \mid Q \in [P]_{\mathcal{D}_{inc}}\}$, $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{inc})$, $Close([P]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{inc}})$, and $Keys([P]_{\mathcal{D}_{upd+}}) = \min\big\{\{K \in Keys([P]_{\mathcal{D}_{inc}}) | K \text{ is not in } \mathcal{D}_{org}\} \bigcup \{K' \cup \{x\} | K' \in Keys([P]_{\mathcal{D}_{inc}}) \wedge K' \text{ is in } \mathcal{D}_{org} \text{ and } x \in Close([P]_{\mathcal{D}_{inc}}) \wedge x \text{ is not in } \mathcal{D}_{org}\}\big\}$.*

3. *P is in $\mathcal{D}_{inc}$, P is in $\mathcal{D}_{org}$, and $f(Q, \mathcal{D}_{inc}) = f(P, \mathcal{D}_{inc})$ for every $Q \in [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where an equivalence class has remained unchanged but with increased support. In this case, $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{inc})$, $Close([P]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}})$ and $Keys([P]_{\mathcal{D}_{upd+}}) = Keys([P]_{\mathcal{D}_{org}})$.*

4. *P is in $\mathcal{D}_{inc}$, P is in $\mathcal{D}_{org}$, and $f(Q, \mathcal{D}_{inc}) \neq f(P, \mathcal{D}_{inc})$ for some $Q \in [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where an equivalence class has split and P is in one of the resulting equivalence classes that has increased in sup-*

port, e.g. EC3 in Figure 4.3. In this case, $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{inc}} \cap [P]_{\mathcal{D}_{org}}$*,* $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{inc})$*,* $Close([P]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}}) \cap Close([P]_{\mathcal{D}_{inc}})$ *and* $Keys([P]_{\mathcal{D}_{upd+}}) = \min\{K \cup K' | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq Close([P]_{\mathcal{D}_{inc}})$ *and* $K' \in Keys([P]_{\mathcal{D}_{inc}}) \wedge K' \subseteq Close([P]_{\mathcal{D}_{org}})\}$*.*

5. *P is not in $\mathcal{D}_{inc}$, P is in $\mathcal{D}_{org}$, and Q is not in $\mathcal{D}_{inc}$ for all $Q \in [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where an equivalence class has remain unchanged in size and in support.*

6. *P is not in $\mathcal{D}_{inc}$, P is in $\mathcal{D}_{org}$, and Q is in $\mathcal{D}_{inc}$ for some $Q \in [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where $[P]_{\mathcal{D}_{org}}$ has to split, as stated in Theorem 4.11, and $[P]_{\mathcal{D}_{upd+}}$ is the unique resulting equivalence that does not have increased in support. In this case,* $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}} - \bigcup\{[Q]_{\mathcal{D}_{inc}} \mid Q \in [P]_{org}\}$*,* $sup(P, \mathcal{D}_{upd+}) = sup(P,\ \mathcal{D}_{org})$*,* $Close([P]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}})$*, and* $Keys([P]_{\mathcal{D}_{upd+}}) = \min\big\{\{K \in Keys([P]_{\mathcal{D}_{org}}) | K \textit{ is not in } \mathcal{D}_{inc}\} \bigcup \{K' \cup \{x\} | K' \in Keys([P]_{\mathcal{D}_{org}}) \wedge K' \textit{ is in } \mathcal{D}_{inc} \textit{ and } x \in Close([P]_{\mathcal{D}_{org}}) \wedge x \textit{ is not in } \mathcal{D}_{inc}\}\big\}$*.*

Proof. Scenario 1 follows from Corollary 4.14. Scenario 2 follows from Corollary 4.15 in a manner that is symmetric to Scenario 6. Scenario 3 follows from Corollary 4.13. Scenario 4 follows from Theorem 4.11. Scenario 5 follows from Corollary 4.12.

The formulas of closed and key patterns (generators) in Scenarios 2, 4 and 6 follow from Definition 2.3 and Fact 4.6. However, the derivation of the key patterns formula in Scenario 4 is not so straight-forward. So we explain it here. In Scenario 4, $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}} \cap [P]_{\mathcal{D}_{inc}}$. Let $\mathcal{L} = \min\{K \cup K' | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq Close([P]_{\mathcal{D}_{inc}})$ *and* $K' \in Keys([P]_{\mathcal{D}_{inc}}) \wedge K' \subseteq Close([P]_{\mathcal{D}_{org}})\}$. We first prove

$Q \in [P]_{\mathcal{D}_{upd+}}$ for every $Q \in \mathcal{L}$. Without loss of generality, assume $Q \in \mathcal{L}$ and $Q = K \cup K'$, where $K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq Close([P]_{\mathcal{D}_{inc}})$ *and* $K' \in Keys([P]_{\mathcal{D}_{inc}}) \wedge K' \subseteq Close([P]_{\mathcal{D}_{org}})$. Since $K \subseteq Close([P]_{\mathcal{D}_{inc}})$ and $K' \subseteq Close([P]_{\mathcal{D}_{inc}})$ (since $K' \in [P]_{\mathcal{D}_{inc}}$), we have $K \cup K' \subseteq Close([P]_{\mathcal{D}_{inc}})$. This implies that $K' \subseteq Q (= K \cup K') \subseteq Close([P]_{\mathcal{D}_{inc}})$ and thus $Q \in [P]_{\mathcal{D}_{inc}}$ (Fact 4.5 & Definition 4.1). With a similar logic, we can show that $Q \in [P]_{\mathcal{D}_{org}}$ as well. Therefore, we have $Q \in [P]_{\mathcal{D}_{upd+}}$ as desired. Next, we prove that $\mathcal{L}$ is the complete set of the most general patterns (key patterns) in $[P]_{\mathcal{D}_{upd+}}$. We assume that there exists $Q \notin \mathcal{L}$ that $Q \in [P]_{\mathcal{D}_{upd+}}$ and Q is most general. $Q \in [P]_{\mathcal{D}_{upd+}}$ implies that $Q \in [P]_{\mathcal{D}_{org}}$ and $Q \in [P]_{\mathcal{D}_{inc}}$ (for $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}} \cap [P]_{\mathcal{D}_{inc}}$). According to Fact 4.5, this implies that $K_1 \subseteq Q \subseteq Close([P]_{\mathcal{D}_{org}})$ and $K_2 \subseteq Q \subseteq Close([P]_{\mathcal{D}_{inc}})$, for some $K_1 \in Keys([P]_{\mathcal{D}_{org}})$ and $K_2 \in Keys([P]_{\mathcal{D}_{inc}})$. This further implies that $K_1 \subseteq Close([P]_{\mathcal{D}_{inc}})$, $K_2 \subseteq Close([P]_{\mathcal{D}_{org}})$ and $Q \supseteq K_1 \cup K_2$. This means that either $Q \in \mathcal{L}$ or $\exists Q' \in \mathcal{L}$ that $Q \supseteq Q'$. This contradicts with the assumption that $Q \notin \mathcal{L}$ and Q is most general in $[P]_{\mathcal{D}_{upd+}}$. Therefore, the assumption is not true, and $\mathcal{L}$ is the complete set of key patterns of $[P]_{\mathcal{D}_{upd+}}$. $\square$

Theorem 4.16 describes how the equivalence classes of the frequent pattern space evolves under incremental updates. Theorem 4.16 enumerates all possible evolution scenarios, and it also summarizes the exact conditions for each evolution scenarios to occur. More importantly, the theorem demonstrates how to update the frequent pattern space based on the equivalence classes in $\mathcal{D}_{inc}$, which is typically smaller and faster to compute. This result has significant implications for the design of efficient algorithms for maintaining frequent pattern spaces and their equivalence classes.

In addition, if the support threshold is defined in terms of percentage, $ms_{\%}$, an incremental update affects the absolute support threshold, ms_a. Recall that $ms_a = \lceil ms_{\%} \times |\mathcal{D}| \rceil$. Since $|\mathcal{D}_{upd+}| > |\mathcal{D}_{org}|$, the updated absolute support threshold $ms'_a = \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil \geq ms_a = \lceil ms_{\%} \times |\mathcal{D}_{org}| \rceil$. Thus, in this case, the absolute support threshold, ms_a, increases after an incremental update. Moreover, this increase in ms_a may cause some existing frequent equivalence classes to become infrequent. $EC2$ in Figure 4.2 is an example.

4.3 Space Evolution under Decremental Updates

We investigate in this section how the frequent pattern space, equivalence classes, and their support values evolve when multiple transactions are removed from an existing dataset. For this section, we use the following notations: $\mathcal{D}_{org}$ is the original dataset, $\mathcal{D}_{dec}$ is the set of old transactions to be removed, and $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \mathcal{D}_{dec}$ is the updated dataset. We assume without loss of generality that $\mathcal{D}_{org} \neq \emptyset$ and $\mathcal{D}_{dec} \subseteq \mathcal{D}_{org}$. Thus we can conclude that, for any pattern P, $f(P, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{org}) - f(P, \mathcal{D}_{dec})$ and $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org}) - sup(P, \mathcal{D}_{dec})$.

There is an obvious duality between incremental updates and decremental updates. In particular, if we first increment a dataset with $\mathcal{D}_{inc}$ and then decrement the result with $\mathcal{D}_{dec} = \mathcal{D}_{inc}$, we get back the original dataset. Conversely, if we first decrement a dataset with $\mathcal{D}_{dec}$ and then increment the result with $\mathcal{D}_{inc} = \mathcal{D}_{dec}$, we get back the original dataset.

Under incremental updates, new equivalence classes may emerge; in contrast,

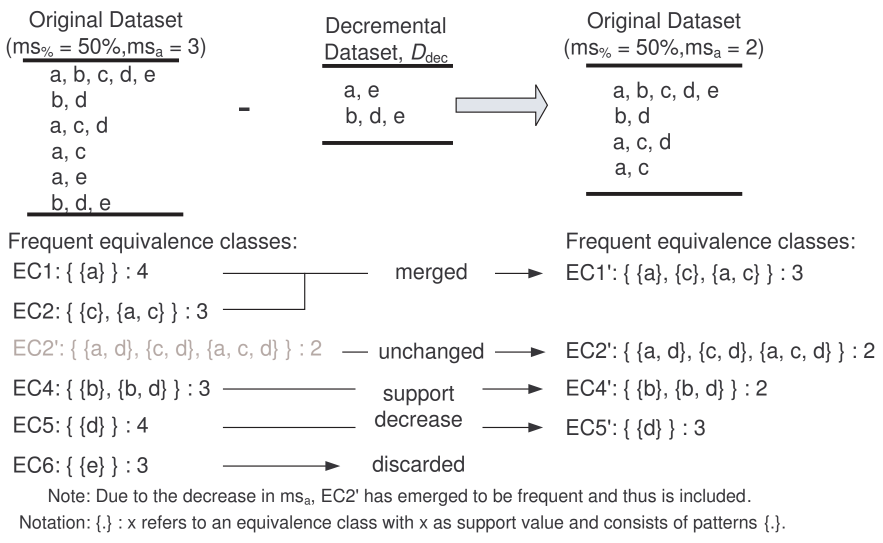

Original Dataset
(ms% = 50%,msa = 3)
a, b, c, d, e
b, d
a, c, d
a, c
a, e
b, d, e
-
Decremental Dataset, Ddec
a, e
b, d, e
Original Dataset
(ms% = 50%,msa = 2)
a, b, c, d, e
b, d
a, c, d
a, c
Frequent equivalence classes:
EC1: { {a} } : 4
EC2: { {c}, {a, c} } : 3
EC2': { {a, d}, {c, d}, {a, c, d} } : 2
EC4: { {b}, {b, d} } : 3
EC5: { {d} } : 4
EC6: { {e} } : 3
merged
unchanged
support decrease
discarded
Frequent equivalence classes:
EC1': { {a}, {c}, {a, c} } : 3
EC2': { {a, d}, {c, d}, {a, c, d} } : 2
EC4': { {b}, {b, d} } : 2
EC5': { {d} } : 3
Note: Due to the decrease in msa, EC2' has emerged to be frequent and thus is included.
Notation: {.} : x refers to an equivalence class with x as support value and consists of patterns {.}.

Figure 4.4: Evolution of equivalence classes under decremental updates.

existing equivalence classes may disappear under decremental updates. Moreover, for those existing equivalence classes that still exist after the decremental update, they may evolve in also three different ways, as demonstrated in Figure 4.4. The first way is to remain unchanged without any change in support. The second way is to remain unchanged but with an decreased support. The third way is to expand—by merging with other classes. We know from Proposition 4.10 that an equivalence class may shrink in size and increase in support after incremental updates. It follows by duality that an equivalence class may increase in size and decrease in support after decremental updates.

Corollary 4.17. *Let P be a pattern in $\mathcal{D}_{upd-}$. Then $[P]_{\mathcal{D}_{upd-}} \supseteq [P]_{\mathcal{D}_{org}}$, and $sup(P, \mathcal{D}_{upd-}) \leq sup(P, \mathcal{D}_{org})$.*

While an equivalence class may be split into multiple classes by an incremental update, multiple classes may be merged by a decremental update. In short, the following dual of Theorem 4.11 holds:

Corollary 4.18. *Let P be a pattern in $\mathcal{D}_{org}$, and Q_1, ..., Q_n be n unique patterns in $\mathcal{D}_{org}$ such that $Q_i \notin [P]_{\mathcal{D}_{org}}$ for $1 \leq i \leq n$ and $[Q_i]_{\mathcal{D}_{org}} \cap [Q_j]_{\mathcal{D}_{org}} = \emptyset$ for $1 \leq i \neq j \leq n$. Then $[P]_{\mathcal{D}_{org}}, [Q_1]_{\mathcal{D}_{org}}, \cdots, [Q_n]_{\mathcal{D}_{org}}$ merge to form $[P]_{\mathcal{D}_{upd-}}$ iff $f(P, \mathcal{D}_{upd-}) = f(Q_i, \mathcal{D}_{upd-})$ for $1 \leq i \leq n$, and n is the largest such integer. Furthermore, $[Q_i]_{\mathcal{D}_{org}} = [Q_i]_{\mathcal{D}_{dec}} \cap [P]_{\mathcal{D}_{upd-}}$ for $1 \leq i \leq n$. Moreover, there is at most one Q_i among Q_1, ..., Q_n, such that Q_i does not occur in $\mathcal{D}_{dec}$ and $Q_i \in [C]_{\mathcal{D}_{org}}$, where C is the closed pattern of $[P]_{\mathcal{D}_{upd-}}$.*

Corollary 4.18 describes the exact conditions for equivalence classes to merge under decremental updates. It also implies that an existing equivalence class will remain structurally unchanged if there is no pattern Q_i satisfying the conditions in Corollary 4.18.

Corollary 4.19. *Let P be a pattern in $\mathcal{D}_{org}$. $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$ iff $f(Q, \mathcal{D}_{upd-}) \neq f(P, \mathcal{D}_{upd-})$ for all $Q \notin [P]_{\mathcal{D}_{org}}$. Furthermore, if P is not in $\mathcal{D}_{dec}$, then $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$. Otherwise, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org}) - sup(P, \mathcal{D}_{dec})$.*

To have a deeper understanding on how the frequent pattern space evolves under decremental updates, we investigate the exact conditions for each evolution scenario to occur. Notation $Close(X)$ denotes the closed pattern of equivalence class X and $Keys(X)$ refers to the generators of X.

Theorem 4.20. *For every equivalence class $[P]_{\mathcal{D}_{org}}$ in $\mathcal{D}_{org}$, exactly one of the 6 scenarios below holds:*

1. P is not in $\mathcal{D}_{dec}$, and $f(P, \mathcal{D}_{org}) \neq f(Q, \mathcal{D}_{upd-})$ for all Q in $\mathcal{D}_{dec}$, cor-

responding to the scenario where an equivalence class has remained totally unchanged after the decremental update. In this case, $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$, $Close([P]_{\mathcal{D}_{upd-}}) = Close([P]_{\mathcal{D}_{org}})$ *and* $Keys([P]_{\mathcal{D}_{upd-}}) = Keys([P]_{\mathcal{D}_{org}})$.

2. P *is not in* $\mathcal{D}_{dec}$, *and* $f(P, \mathcal{D}_{org}) = f(Q, \mathcal{D}_{upd-})$ *for some* Q *occurring in* $\mathcal{D}_{dec}$, *corresponding to the scenario where the equivalence class of* Q *has to be merged into the equivalence class of* P. *In this case, let all such* Q*'s in* $\mathcal{D}_{dec}$ *be grouped into* n *distinct equivalence classes* $[Q_1]_{\mathcal{D}_{dec}}, \cdots, [Q_n]_{\mathcal{D}_{dec}}$, *having representatives* $Q_1, \cdots, Q_n$ *satisfying the condition on* Q. *Then* $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup \bigcup_{1 \leq i \leq n} [Q_i]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$, $Close([P]_{\mathcal{D}_{upd-}}) = Close([P]_{\mathcal{D}_{org}})$ *and* $Keys([P]_{\mathcal{D}_{upd-}}) = \min\{K | K \in Keys([P]_{\mathcal{D}_{org}})$ *or* $K \in Keys([Q_i]_{\mathcal{D}_{org}})$ *for* $1 \leq i \leq n\}$. *Furthermore,* $[Q_i]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{upd-}}$ *for* $1 \leq i \leq n$.

3. P *is in* $\mathcal{D}_{dec}$, *and* $f(P, \mathcal{D}_{org}) = f(P, \mathcal{D}_{dec})$, *corresponding to the scenario where the equivalence class has disappeared.*

4. P *is in* $\mathcal{D}_{dec}$, $f(P, \mathcal{D}_{org}) \neq f(P, \mathcal{D}_{dec})$ *and* $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{org})$ *for some* Q *in* $\mathcal{D}_{org}$ *but not in* $\mathcal{D}_{dec}$, *corresponding to the scenario where the equivalence class of* P *has to be merged into the equivalence class of* Q. *This scenario is complement to Scenario 2. In this case, the equivalence class, support, closed pattern and generators of* $[P]_{\mathcal{D}_{upd-}}$ *is same as that of* $[Q]_{\mathcal{D}_{upd-}}$, *as computed in Scenario 2.*

5. *P is in $\mathcal{D}_{dec}$, $f(P,\mathcal{D}_{org}) \neq f(P,\mathcal{D}_{dec})$, $f(P,\mathcal{D}_{upd-}) \neq f(Q,\mathcal{D}_{upd-})$ for all $Q \notin [P]_{\mathcal{D}_{org}}$, corresponding to the situation where the equivalence class has remained unchanged but has decreased in support. In this case, $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$, $sup(P,\mathcal{D}_{upd-}) = sup(P,\mathcal{D}_{org}) - sup(P,\mathcal{D}_{dec})$, $Close([P]_{\mathcal{D}_{upd-}}) = Close([P]_{\mathcal{D}_{org}})$ and $Keys([P]_{\mathcal{D}_{upd-}}) = Keys([P]_{\mathcal{D}_{org}})$.*

6. *P is in $\mathcal{D}_{dec}$, $f(P,\mathcal{D}_{org}) \neq f(P,\mathcal{D}_{dec})$, and $f(P,\mathcal{D}_{upd-}) = f(Q,\mathcal{D}_{upd-})$ for some Q in both $\mathcal{D}_{org}$ and $\mathcal{D}_{dec}$ but $Q \notin [P]_{\mathcal{D}_{org}}$, corresponding to the situation where the equivalence classes of P and Q have to be merged. In this case, let all such Q's in $\mathcal{D}_{org}$ be grouped into n distinct equivalence classes $[Q_1]_{\mathcal{D}_{org}}$, ..., $[Q_n]_{\mathcal{D}_{org}}$, having representatives Q_1, ..., Q_n satisfying the condition on Q. Then $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup \bigcup_{1\leq i\leq n}[Q_i]_{\mathcal{D}_{org}}$, $sup(P,\mathcal{D}_{upd-}) = sup(P,\mathcal{D}_{org}) - sup(P, \mathcal{D}_{dec})$. $Close([P]_{\mathcal{D}_{upd-}}) = \max\{C | C = Close([P]_{\mathcal{D}_{org}})$ or $C = Close([Q_i]_{\mathcal{D}_{org}})$ for $1 \leq i \leq n\}$ and $Keys([P]_{\mathcal{D}_{upd-}}) = \min\{K | K \in Keys([P]_{\mathcal{D}_{org}})$ or $K \in Keys([Q_i]_{\mathcal{D}_{org}})$ for $1 \leq i \leq n\}$. Furthermore, $[Q_i]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{upd-}}$ for $1 \leq i \leq n$.*

Proof. Scenario 3 is obvious. Scenario 1 and Scenario 5 follow from Corollary 4.19, and Scenarios 2, 4 and 6 follow from Corollary 4.18.

Moreover, the formulas of closed and generators in Scenario 2 and Scenario 6 extend from Definition 2.3 and Fact 4.6. □

Theorem 4.20 summarizes all possible scenarios on how the frequent pattern space can evolve after a decremental update. The theorem also describes how the updated frequent equivalence classes in $\mathcal{D}_{upd-}$ can be derived from the existing frequent equivalence classes of $\mathcal{D}_{org}$. Similar to Theorem 4.16, Theorem 4.20 lays a theoretical foundation for the development of effective decremental maintenance algorithms.

In addition, opposite to the incremental update, the decremental update decreases the absolute support threshold if the support threshold is initially defined in terms of percentage. Let the original absolute support $ms_a = \lceil ms_{\%} \times |\mathcal{D}_{org}| \rceil$. Since $|\mathcal{D}_{upd-}| = |\mathcal{D}_{org}| - |\mathcal{D}_{dec}|$, the updated absolute support threshold $ms'_a = \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil < ms_a$. This decrease in the absolute support threshold induces new frequent equivalence classes to emerge, e.g. $EC2'$ in Figure 4.4.

4.4 Space Evolution under Threshold Adjustments

This section discusses how the space of frequent patterns and equivalence classes evolve when the minimum support threshold is adjusted. We use the following notations: $\mathcal{D}$ is the dataset, ms_{org} is the original support threshold, ms_{upd} is the updated support threshold. The minimum support threshold can be defined in terms of either percentage or absolute count, and they are interchangeable via formula $ms_a = \lceil ms_{\%} \times |\mathcal{D}| \rceil$. For simplicity of discussion, we assume here the support threshold is defined in terms of absolute count. We further assume without loss of generality that $\mathcal{D}$ and $|\mathcal{D}|$ remain unchanged for support threshold adjustments.

Under threshold adjustments, since the dataset $\mathcal{D}$ remains unchanged, most properties of patterns are preserved. In particular, all closed and key (generator) patterns remain to be closed and key, and, conversely, all non-closed and non-key patterns remain to be non-closed and non-key, meaning that no new closed and key patterns will emerge. In addition, the support values of all patterns remain unchange. As a result, all equivalence classes in the pattern space will remain structurally unchanged, and their support values will also remain the same.

Support threshold adjustment, however, may invalidate existing frequent patterns and their equivalence classes and may also induce new frequent patterns and equivalence classes to emerge. When the support threshold is adjusted up, existing frequent patterns and equivalence classes may become infrequent. In this case, the frequent pattern space shrinks. The updated frequent pattern space can be obtained by removing existing equivalence classes that are no longer frequent. On the other hand, when the support threshold is adjusted down, new frequent patterns and equivalence classes may emerge. In this case, the frequent pattern space expands. The updated pattern space can be obtained by including newly frequent equivalence classes.

Proposition 4.21. *Let $\mathcal{D}$ be the dataset, ms_{org} be the original minimum support threshold and ms_{upd} be the updated support threshold. Then the updated frequent pattern space $\mathcal{F}(\mathcal{D}, ms_{upd}) = \{P | sup(P, \mathcal{D}) \geq ms_{upd}\}$. Furthermore, if $ms_{upd} \geq ms_{org}$, $\mathcal{F}(\mathcal{D}, ms_{upd}) \subseteq \mathcal{F}(\mathcal{D}, ms_{org})$; and if $ms_{upd} \leq ms_{org}$, $\mathcal{F}(\mathcal{D}, ms_{upd}) \supseteq \mathcal{F}(\mathcal{D}, ms_{org})$*

Proof. This proposition follows from the definition of frequent pattern space (Section 2.1). □

4.5 Summary

This chapter has analyzed the evolution of the frequent pattern space under incremental updates, decremental updates and support threshold adjustments. Since the pattern space is too large to be analyzed directly, we proposed to structurally decompose the pattern space into convex sub-spaces — equivalence classes. The evolution of the frequent pattern space is then studied based on the concept of equivalence classes. We demonstrated that: for incremental and decremental updates, equivalence classes of the frequent pattern space can be updated based on equivalence classes in the incremental or decremental dataset, which are much smaller in terms of number and much faster to compute in terms of time; and, for support threshold adjustment, the updated frequent pattern space can be obtained based on existing frequent equivalence classes. These results lay a theoretical foundation for the development of effective maintenance algorithms.

Part III

ALGORITHMS

Chapter 5

Transaction Removal Update Maintainer (**TRUM**): A Decremental Maintainer

Decremental updates are one of the most common operations in Data-Base Management Systems (*DBMS*) [76]. Decremental updates allow data users to remove obsolete transactions that are no longer in interest and to remove incorrect transactions. In addition, decremental updates are also necessary for hypothesis test and retrospective trend analysis. Therefore, effective decremental maintenance algorithms are required to update the frequent pattern space. However, in the literature, decremental maintenance has not received as much research attention as incremental maintenance. Only a few incremental maintenance algorithms, such as FUP2H, ZIGZAG and moment, are generalized to address also decremental maintenance.

In this chapter, we propose a novel decremental maintenance algorithm to update the frequent pattern space. The proposed algorithm is named as **Transaction Removal Update Maintainer** (**TRUM**). TRUM is developed based on the space evolution analysis in Chapter 4. TRUM maintains the pattern space effectively by updating only the affected equivalence classes. Furthermore, we introduce a new data structure — “Transaction-ID Tree” (*TID-tree*) — to facilitate the maintenance operations involved in TRUM.

5.1 Transaction Removal Update Maintainer (TRUM)

In this section, inspired by the space evolution analysis, we propose a novel algorithm to effective maintain the frequent pattern space for decremental updates.

Recall that, we can structurally decompose the frequent pattern space into convex equivalence classes. This structural decomposition of the pattern space allows us to address the pattern space maintenance in a divide-and-conquer manner. In particular, we can maintain the space of frequent patterns by maintaining the frequent equivalence classes. Moreover, equivalence classes can be concisely represented by their borders — the corresponding closed patterns and generators. This allows us to further simply the maintenance problem to the update of equivalence class borders.

According to the space evolution analysis, in particular Proposition 4.17, the size of equivalence classes may grow after decremental updates, and the support of equivalence classes will decrease. This implies that some existing frequent equivalence classes may become no longer frequent after decremental updates. To be more precise, existing (frequent) equivalence classes may evolve in three different ways under decremental updates: one existing equivalence class may remain structurally unchanged and has the same support; or one existing equivalence class may remain structurally unchanged but with a decreased support; or one existing equivalence class may expand by merging with other equivalence classes. Exact conditions for each evolution scenario are illustrated in Theorem 4.20.

Based on Proposition 4.17 and Theorem 4.20, we summarize the decremental maintenance of frequent pattern space into 3 major computational tasks. The first task is to update the support of existing frequent equivalence classes. After decremental updates, it is necessary to update the support values of all "involved" equivalence classes. The "involved" equivalence classes are those that satisfy scenarios 2 to 5 in Theorem 4.20. We name this computational task the ***Support Update*** task. The second task is to merge equivalence classes that satisfy scenarios 2, 4 and 6 in Theorem 4.20. We name this task the ***Class Merging*** task. Since some existing frequent equivalence classes may become infrequent after the update, the third task is to remove these equivalence classes from the pattern space. We name this task the ***Class Removal*** task. To address these 3 computational tasks, we proposed an effective algorithm.

Algorithm 1 TRUM

```
Input: $\mathcal{D}_{dec}$, the decremental dataset; $\mathcal{F}_{org}$, the frequent pattern space of $\mathcal{D}_{org}$ represented
    with borders of equivalence classes; $ms_a$, absolute count minimum support threshold.
Output: $\mathcal{F}_{upd-}$, the updated frequent pattern space.
Method:
 1: $\mathcal{F}_{upd-} := \mathcal{F}_{org}$; {Initialization.}
 2: for all transaction $T \in \mathcal{D}_{dec}$, and all equivalence classes $EC \in \mathcal{F}_{upd-}$ do
 3:    if $EC.closed \subseteq T$ then
 4:       $EC.support := EC.support - 1$;
          {Update the support of existing frequent equivalence classes.}
 5:    end if
 6: end for
 7: for all $EC \in \mathcal{F}_{upd-}$ do
 8:    if $EC.support < ms_a$ then
 9:       Remove $EC$ from $\mathcal{F}_{upd-}$, continue;
          {Remove existing equivalence classes that are no longer frequent.}
10:    end if
11:    for all $EC' \in \mathcal{F}_{upd-}$ that $EC' \neq EC$ do
12:       if $f(EC', \mathcal{D}_{upd-}) = f(EC, \mathcal{D}_{upd-})$ & $EC'.closed \subset EC.closed$ then
13:          $EC.keys := min\{K | K \in EC.keys\ or\ K \in EC'.keys\}$
14:          Remove $EC'$ from $\mathcal{F}_{upd-}$, continue;
             {Merge $EC'$ into $EC$.}
15:       end if
16:       if $f(EC', \mathcal{D}_{upd-}) = f(EC, \mathcal{D}_{upd-})$ & $EC'.closed \supset EC.closed$ then
17:          $EC'.keys := min\{K | K \in EC.keys\ or\ K \in EC'.keys\}$
18:          Remove $EC$ from $\mathcal{F}_{upd-}$, break;
             {Merge $EC$ into $EC'$.}
19:       end if
20:    end for
21: end for
    return $\mathcal{F}_{upd-}$;
```

The proposed algorithm is called the "Transaction Removal Update Maintainer", in short TRUM. TRUM effectively maintains the frequent pattern space by updating the borders of equivalence classes instead of the entire pattern space. The pseudo-code of TRUM is presented in Algorithm 1. In TRUM, we assume that the minimum support threshold is defined in terms of absolute counts. Absolute count threshold is often applied in bioinfomatic applications, such as structural motif finding [71]. In the case, where absolute count threshold is used, the support threshold remains constant after decremental updates. In Algorithm 1, we use notations $X.closed$ to denote the closed pattern of an equivalence class, $X.keys$ to denote the set of generators (key patterns) of an equivalence class, $X.support$ to denote the support value of an equivalence class, and $f(.,.)$ to denote the filter defined in Definition 4.3.

Theorem 5.1. *TRUM, given in Algorithm 1, maintains the borders and support values of equivalence classes correctly and completely for decremental updates.*

Proof. According to Theorem 4.20, for any equivalence class $[P]_{\mathcal{D}_{org}}$ in $\mathcal{D}_{org}$, there are only 6 possible scenarios . We prove the correctness and completeness of the proposed algorithm according to these 6 scenarios.

For Scenario 1, suppose (i) P is frequent in $\mathcal{D}_{org}$, (ii) P is not in $\mathcal{D}_{dec}$, and (iii) $f(P, \mathcal{D}_{org}) \neq f(Q, \mathcal{D}_{upd-})$ for all Q in $\mathcal{D}_{dec}$. Let EC_P denote the equivalence class of P in $\mathcal{F}_{org}$, the original pattern space. During initialization, EC_P is included into $\mathcal{F}_{upd-}$ by Line 1 as default. Point (ii) implies $EC_P.closed$ is not in any transaction of $\mathcal{D}_{dec}$. Thus Line 3 cannot hold on EC_P. Therefore, the support of EC_P remains unchanged as desired. Point (i) further implies $EC_P.support \geq ms_a$. Thus Line 8 cannot hold, and Lines 9 and 10 are skipped as desired. Next, we consider those $EC' \in \mathcal{F}_{upd-}$, where $EC' \neq EC_P$. Point (iii) directly implies that Line 12 and Line 16 cannot hold. Therefore, EC_P remains totally unchanged as desired. This proves Scenario 1.

For Scenario 2, suppose (i) P is frequent in $\mathcal{D}_{org}$, (ii) P is not in $\mathcal{D}_{dec}$, and (iii) $f(P, \mathcal{D}_{org}) = f(Q, \mathcal{D}_{upd-})$ for some Q in $\mathcal{D}_{dec}$. Let EC_P denote the equivalence class of P in $\mathcal{F}_{org}$. Similar to Scenario 1, EC_P is included into $\mathcal{F}_{upd-}$ by Line 1 as default. Point (ii) implies $EC_P.closed$ is not in any transaction of $\mathcal{D}_{dec}$. Thus Line 3 cannot hold on EC_P. Therefore, the support of EC_P remains unchanged as desired. Also, Point (i) further implies $EC_P.support \geq ms_a$. Thus Line 8 cannot hold, and Lines 9 and 10 are skipped as desired.

Next, we consider those $EC' \in \mathcal{F}_{upd-}$, where $EC' \neq EC_P$. First, we show that, for each EC', Line 16 cannot hold. To prove this, we first assume that there exists EC' that satisfies Line 16. For this particular EC', we have (1) $f(EC', \mathcal{D}_{upd-}) = f(EC_P, \mathcal{D}_{upd-})$ and (2) $EC'.closed \supset EC_P.closed$. Point (iii) implies that $EC'.closed$ is in $\mathcal{D}_{dec}$. Then since $EC'.closed \supset EC_P.closed$ (Point (2)), we have $EC_P.closed$ is in $\mathcal{D}_{dec}$, which means P is also in $\mathcal{D}_{dec}$. This contradicts with Point (ii), and thus our assumption is not valid. As desired, Line 16 cannot hold for all EC' in this case.

Second, we show that Line 12 holds for some EC'. Point (iii) implies that there exists $Q \notin EC_P$ such that $f(EC_P, \mathcal{D}_{org}) = f(Q, \mathcal{D}_{upd-})$ and Q is in $\mathcal{D}_{dec}$. Let EC_Q denote the equivalence class of Q in $\mathcal{F}_{org}$. Then we have: (1) $EC_Q \neq EC_P$, (2) $f(EC_P, \mathcal{D}_{org}) = f(EC_Q, \mathcal{D}_{upd-})$ and (3) patterns of EC_Q, including $EC_Q.closed$, are in $\mathcal{D}_{dec}$. Point (2) directly implies the first condition in Line 12 is satisfied. For the second condition of Line 12, Point (3) implies that $f(EC_Q.closed, \mathcal{D}_{upd-}) = f(EC_Q.closed, \mathcal{D}_{org}) - f(EC_Q.closed, \mathcal{D}_{dec})$. Also since $f(EC_P, \mathcal{D}_{org}) = f(EC_Q, \mathcal{D}_{upd-})$, we have $f(EC_P.closed, \mathcal{D}_{org}) = f(EC_Q.closed, \mathcal{D}_{org}) - f(EC_Q.closed, \mathcal{D}_{dec})$. This further implies $f(EC_P.closed, \mathcal{D}_{org}) \subset f(EC_Q.closed, \mathcal{D}_{org})$. This means $EC_P.closed \supset EC_Q.closed$ (Fact 4.8). With this, we show that $\exists EC_Q$ such that $EC_Q \neq EC_P$ and EC_Q satisfies Line 12. All such EC_Qs will be merged into EC_P by Lines 13 and 14 as desired. This proves Scenario 2.

For Scenario 3, suppose (i) P is frequent in $\mathcal{D}_{org}$, (ii) P is in $\mathcal{D}_{dec}$, and (iii) $|f(P, \mathcal{D}_{upd-})| < ms_a$. Let EC_P denote the equivalence class of P in $\mathcal{F}_{org}$. EC_P is included into $\mathcal{F}_{upd-}$ by Line 1 as default. Point (ii) implies that $\exists T \in \mathcal{D}_{dec}$ such that $EC_P.closed \subseteq T$. Thus Line 3 will be satisfied for such Ts, and $EC_P.support$ is iteratively updated by Line 4. Next, Point (iii) directly implies that Line 8 will be satisfied. Therefore,

EC_P is correctly removed from the pattern space as described in Scenario 3.

Scenario 4 is complementary to Scenario 2. So we do not repeat the proof here.

For Scenario 5, suppose (i) P is frequent in $\mathcal{D}_{org}$, (ii) P is in $\mathcal{D}_{dec}$, (iii) $|f(P, \mathcal{D}_{upd-})| \geq ms_a$, (iv) $f(P, \mathcal{D}_{upd-}) \neq f(Q, \mathcal{D}_{upd-})$ for all $Q \notin [P]_{\mathcal{D}_{org}}$. Let EC_P denote $[P]_{\mathcal{D}_{org}}$. EC_P is included into $\mathcal{F}_{upd-}$ by Line 1 as default. Point (ii) implies that $\exists T \in \mathcal{D}_{dec}$ such that $EC_P.closed \subseteq T$. Thus Line 3 will be satisfied for such Ts, and $EC_P.support$ is iteratively updated by Line 4. Point (iii) then directly implies that Line 8 will not be satisfied. Thus Lines 9 and 10 are skipped. Next, Point (iv) ensures that Lines 12 and 16 will not be satisfied for all $EC' \neq EC_P$. Therefore, EC_P remains unchanged but with a decreased support as desired.

For Scenario 6, suppose (i) P is frequent in $\mathcal{D}_{org}$, (ii) P is in $\mathcal{D}_{dec}$, (iii) $|f(P, \mathcal{D}_{upd-})| \geq ms_a$, (iv) $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$ for some Q in both $\mathcal{D}_{org}$ and $\mathcal{D}_{dec}$ but $Q \notin [P]_{\mathcal{D}_{org}}$. Let EC_P denote the equivalence class of P in $\mathcal{F}_{org}$. EC_P is included into $\mathcal{F}_{upd-}$ by Line 1 as default. Point (ii) implies that $\exists T \in \mathcal{D}_{dec}$ such that $EC_P.closed \subseteq T$. Thus Line 3 will be satisfied for such Ts, and $EC_P.support$ is iteratively updated by Line 4. Point (iii) then directly implies that Line 8 will not be satisfied. Thus Lines 9 and 10 are skipped. Next, Point (iv) means that Line 12 and 16 will be satisfied for some $EC' \neq EC_P$. All such EC' will merge with EC_P by Lines 13 & 14 or Lines 17 & 18 as desired. Thus equivalence classes under Scenario 6 are correctly maintained. This completes the proof. $\square$

We have demonstrated in Theorem 5.1 the correctness and completeness of our proposed algorithm TRUM. We now investigate the computational complexity of TRUM. As shown in Algorithm 1, TRUM addresses all three computational tasks of decremental maintenance. The *Support Update* task is handled by Lines 3 to 5 in

Algorithm 1. According to Algorithm 1, the straightforward approach for the *Support Update* task is to have a full scan of all existing frequent equivalence classes for each transaction T in the decremental dataset $\mathcal{D}_{dec}$. In this approach, the computational complexity to accomplish this task is $O(N_{EC} \times |\mathcal{D}_{dec}|)$, where N_{EC} refers to the number of existing frequent equivalence classes. The *Class Merging* task is handled by Lines 11 to 20. For this task, the most straightforward approach is to have a pairwise comparison among all existing equivalence classes. This pairwise comparison is computationally expensive, and its computational complexity is $O(N_{EC}^2)$. The *Class Removal* task is handled by Lines 8 to 10. This task is comparatively simpler. It requires only one scan of the existing equivalence classes, and its computational complexity is $O(N_{EC})$.

We observe that the *Support Update* and *Class Merging* tasks are the major contributors to the computational costs of TRUM. The complexity of the straightforward approaches for these two tasks is $O(N_{EC} \times |\mathcal{D}_{dec}|)$ and $O(N_{EC}^2)$ respectively. Although N_{EC}, the number of existing frequent equivalence classes, is much smaller than the number of frequent patterns, N_{EC} is still very large in general. Therefore, the *Support Update* and *Class Merging* tasks are computational challenging. Effective techniques are required to complete these two tasks. To address this, we develop a novel maintenance data structure. We name the data structure “Transaction-ID Tree”, in short *TID-tree*.

5.2 Maintenance Data Structure: Transaction-ID Tree (*TID-tree*)

Transaction-ID Tree (*TID-tree*) is proposed to facilitate the computational tasks in TRUM. *TID-tree* is developed based on the concept of Transaction Identifier List, in short *TID-list*. Thus we first recap the basic properties of *TID-list*.

TID-list is a very popular concept in the literature of frequent pattern mining [27, 66, 67, 70]. For a pattern P, its *TID-list* is a list of IDs of those transactions that contain pattern P. *TID-list*s, serve as the vertical projections of items, greatly facilitate the discovery of frequent patterns and the counting of their support. In our proposed data structure *TID-tree*, a new feature of *TID-list* is exploited. *TID-list*s are utilized as identifers of equivalence classes. As shown in Figure 5.1 (b), each frequent equivalence class in the pattern space is associated with a *TID-list*. In this case, the *TID-list* records the transactions that the corresponding equivalence class appears in. To construct *TID-list*s for equivalence classes, we need to assign a unique "Transaction ID" (*TID*) to each transaction as shown in Figure 5.1 (a). According to the definition of equivalence class (Definition 4.3), each equivalence class corresponds to a unique *TID-list* and so can be identified by it. This observation inspires the development of *TID-tree*.

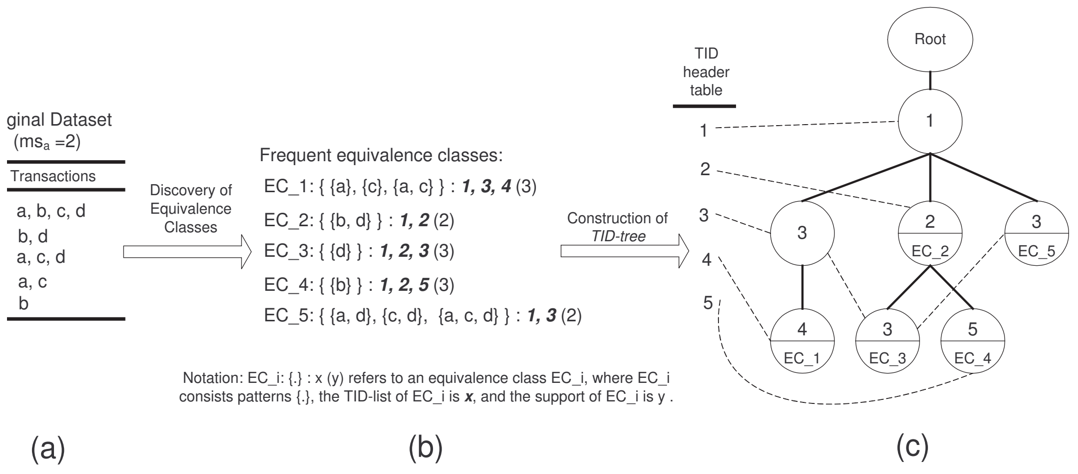

ginal Dataset
(ms_a =2)
Transactions
a, b, c, d
b, d
a, c, d
a, c
b
Discovery of Equivalence Classes
Frequent equivalence classes:
EC_1: { {a}, {c}, {a, c} } : ***1, 3, 4*** (3)
EC_2: { {b, d} } : ***1, 2*** (2)
EC_3: { {d} } : ***1, 2, 3*** (3)
EC_4: { {b} } : ***1, 2, 5*** (3)
EC_5: { {a, d}, {c, d}, {a, c, d} } : ***1, 3*** (2)
Notation: EC_i: {.} : x (y) refers to an equivalence class EC_i, where EC_i consists patterns {.}, the TID-list of EC_i is ***x***, and the support of EC_i is y .
Construction of *TID-tree*
TID header table
1
2
3
4
5
Root
1
3
2
EC_2
3
EC_5
4
EC_1
3
EC_3
5
EC_4
(a)
(b)
(c)

Figure 5.1: An example of *TID-tree*. (a) The original dataset; (b) the frequent equivalence classes of original dataset when $ms_a = 2$; and (c) the corresponding *TID-tree*.

TID-tree is a prefix tree of *TID-list*s of equivalence classes. Prefix trees have been widely used as concise storages of frequent patterns [33] and closed patterns [28]. In TRUM, *TID-tree* serves as a concise storage of *TID-list*s of the existing frequent equivalence classes. Figure 5.1 (c) shows how *TID-list*s of frequent equivalence classes in (b) can be compressed into a *TID-tree*. Details of the construction of a prefix tree can be referred to [28]. Here, we emphasize two features of *TID-tree*: (1) Each node in the *TID-tree* stores a *TID*. If *TID* of the node is the last *TID* of an equivalence class's *TID-list*, the node then points to the corresponding equivalence class. Moreover, the depth of the node reflects the support of the corresponding equivalence class. For example, equivalence class *EC*_1 in Figure 5.1 is associated with the leaf node of *TID-tree*, whose *TID* value is 4. The depth of this particular leaf node is 3, and thus the support of *EC*_1 is 3. (2) *TID-tree* has a header table, where each slot stores a linked list that connects all the nodes with the same *TID*. This header table allows us to efficiently retrieve nodes with identical *TID*s.

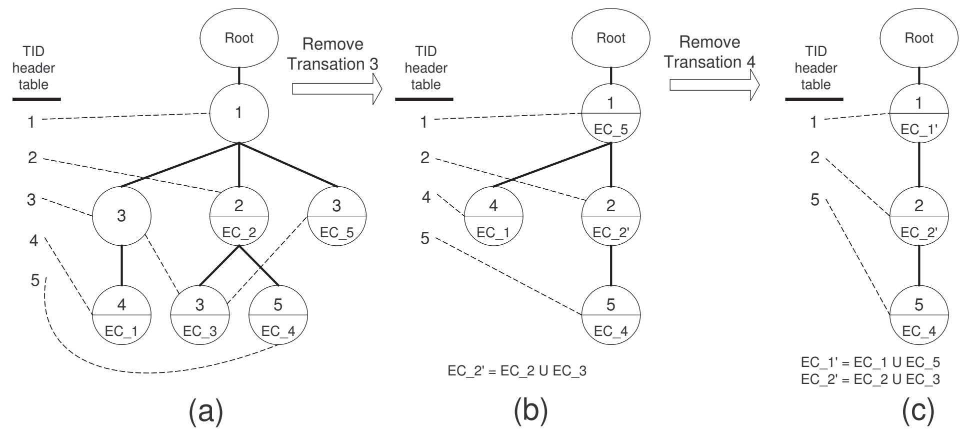


Figure 5.2: Update of *TID-tree*. (a) The original *TID-tree*; (b) the updated *TID-tree* after removing transaction 3; and (c) the updated *TID-tree* after removing also transaction 4.

Figure 5.1 demonstrates that *TID-tree* is a compact data structure that records existing equivalence classes, their support values and their *TID-list*s. The next question is: how *TID-tree* can be employed to facilitate the maintenance computational tasks? How *TID-tree* can be efficiently updated under decremental updates?

When transactions are removed from the original dataset, the corresponding *TID-tree* is updated by removing all the nodes that include the *TID*s of deleted transactions. This can be accomplished effectively with the help of the *TID* header table. As demonstrated in Figure 5.2, after a node is removed, its children re-link to its parent to maintain the tree structure. If the node points to an equivalence class, the pointer is passed to its parent. When two or more equivalence class pointers collide into one node, according to the definition of *TID-tree*, it means that they appear in the same set of transactions after the update. Therefore, these equivalence classes will be merge together. This effectively addresses the *Class Merging* task of TRUM. In Figure 5.2, equivalence classes EC_2 and EC_3 of the original dataset merge into EC_2' after removing transaction 3, and EC_1 and EC_5 merge into EC_1' after removing also transaction 4. In addition, since the depth of an equivalence class node in the *TID-tree* indicates the support of the class, updating the *TID-tree* simultaneously updates the support of equivalence classes. This conveniently solves the *Support Update* task.

As illustrated in Figure 5.2, *TID-tree* allows TRUM to effectively maintain the frequent pattern space by updating only the equivalence classes that are affected by the update. Recall that, with the straightforward approaches, the computational complexity of *Support Update* and *Class Merging* tasks is $O(N_{EC} \times |\mathcal{D}_{dec}|)$ and $O(N_{EC}^2)$ respectively, where N_{EC} refers to the number of existing frequent equivalence classes and $|\mathcal{D}_{dec}|$ denotes the size of the decremental dataset. With *TID-tree*, these two maintenance tasks of TRUM are accomplished in one step as we remove *TID*s from the *TID-tree*. The *TID* linked lists need to be removed one by one. Therefore, the computational complexity of these two maintenance tasks is significantly reduced to $O(|\mathcal{D}_{dec}|)$.

Based on this complexity analysis, we conclude that: with *TID-tree*, TRUM is much more computationally effective compared to previous works, such as [27, 70], whose computational complexity is $O(N_{FP})$, where N_{FP} refers to the number of frequent patterns. This is because $|\mathcal{D}_{dec}| \ll N_{FP}$.

5.3 Experimental Studies

Extensive experiments were performed to evaluate the efficiency of TRUM. Experiments were conducted with the benchmark datasets from the *FIMI* Repository [25].

Dataset	Size	#Trans	#Items	maxTL	aveTL
accidents	34.68MB	340,183	468	52	33.81
BMS-POS	11.62MB	515,597	1,657	165	6.53
BMS-WEBVIEW-1	0.99MB	59,602	497	268	2.51
BMS-WEBVIEW-2	2.34MB	77,513	3,340	162	4.62
chess	0.34MB	3,196	75	37	37.00
connect-4	9.11MB	67,557	129	43	43.00
mushroom	0.56MB	8,124	119	23	23.00
pumsb	16.30MB	49,046	2,113	74	74.00
pumsb_star	11.03MB	49,046	2,088	63	50.48
retail	4.07MB	88,162	16,470	77	10.31
T10I4D100K	3.93MB	100,000	870	30	10.10
T40I10D100K	15.13MB	100,000	942	78	39.61

Table 5.1: Characteristics of testing datasets [49]. Notations: #Trans denotes the total number of transactions in the dataset, #Items denotes the total number of distinct items, maxTL denotes the maximal transaction length and aveTL is the average transaction length.

The statistical information of the benchmark datasets is summarized in Table 5.1. The benchmark datasets include both real-world and synthetic datasets. Experiments were run on a PC with 2.4GHz CPU and 3.2G of memory.

TRUM is tested under various updated intervals. Here, the update interval is defined as $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$. In applications of database management and interactive mining, the size of decremental dataset $\mathcal{D}_{dec}$, meaning the number of transactions to be removed, is generally much smaller than the size of original dataset $\mathcal{D}_{org}$. It is not common for a data user to remove more than 10% of data at one single update. Therefore, tests were run for $\Delta^- \leq 10\%$. TRUM is also tested under various minimum support thresholds. In addition, we observe that the performance of the algorithm varies slightly when different sets of transactions are removed. To have a stable performance measure, for each update interval, 5 random sets of transactions were removed, and the average performance of the algorithm was recorded. **NOTE:** this averaging strategy is applied in all experimental studies of the Thesis.

To justify the effectiveness of TRUM, the performance of TRUM is compared with both frequent pattern discovery and maintenance algorithms.

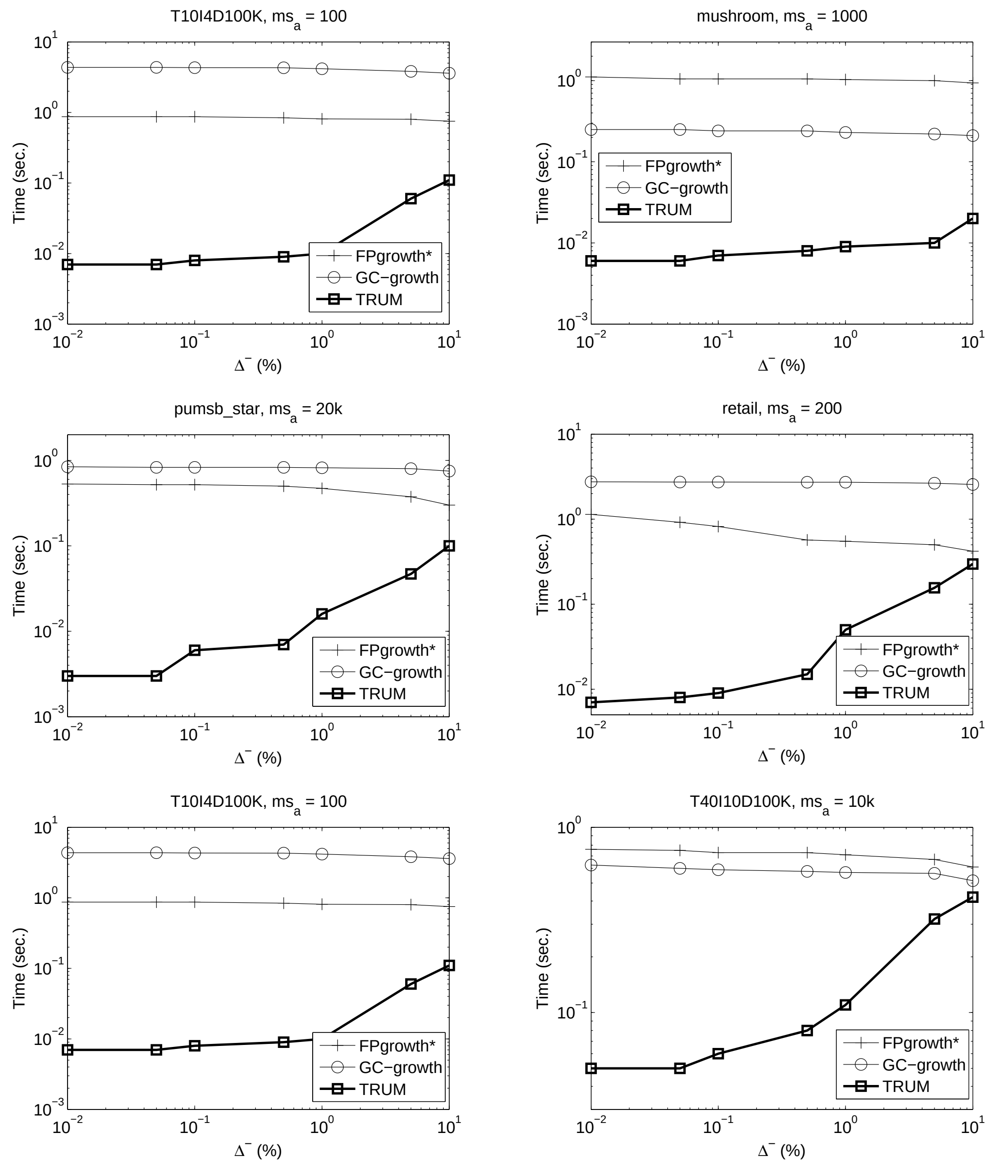


Figure 5.3: Performance comparison of TRUM and the pattern discovery algorithms, FPgrowth* and GC-growth, under batch maintenance. Notations: $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$.

5.3.1 Comparison with Discovery Algorithms

TRUM is compared with state-of-the-art discovery algorithm FPgrowth* [28], which is the fastest implementation of FP-growth [33]. TRUM is also compared with the recently proposed algorithm GC-growth [47], which is the fastest discovery algorithm for frequent equivalence classes. The effectiveness of TRUM is compared with the discovery algorithms under the settings of both batch maintenance, which is common in data management, and eager maintenance, which is necessary for some interactive mining applications, e.g. trend analysis.

Batch Maintenance

In batch maintenance, decremental transactions in $\mathcal{D}_{dec}$ are first removed, as a batch, from the original dataset $\mathcal{D}_{org}$ to obtain the updated dataset $\mathcal{D}_{upd-}$. Then, discovery algorithms are applied to re-discover the updated frequent pattern space based on $\mathcal{D}_{upd-}$.

Figure 5.3 graphically compares the batch maintenance performance of TRUM with the discovery algorithms FPgrowth* and GC-growth. It can be seen that, compared with re-generating the frequent pattern space with the most effective discovery algorithms, it is more effective to maintain the pattern space with TRUM. The average computational "speed up" of TRUM compared with the discovery algorithms is summarized in Table 5.2. The computational speed up is calculated as:[1]

$$SpeedUp(Algo_{target}, Algo_{compare}) = T_{compare}/T_{target} \tag{5.1}$$

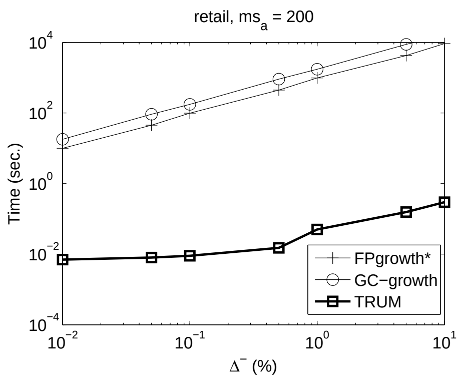
where $SpeedUp(Algo_{target}, Algo_{compare})$ refers the speed up achieved by the target

[1]This speed up formula is used through out the Thesis

algorithm, $Algo_{target}$, against the comparing algorithm, $Algo_{compare}$; and T_{target} and $T_{compare}$ denotes the execution time of the target algorithm and comparing algorithm respectively. Here, the target algorithm is TRUM. As shown in Table 5.2, TRUM outperforms FPgrowth*, on average, 2 orders of magnitude. For the best scenarios, e.g. dataset *mushroom*, TRUM runs faster than FPgrowth* by more than 3 orders of magnitude. For the worst scenarios, like dataset *T40I10D100K*, TRUM is still 5 times faster than FPgrowth*. Compared with GC-growth, the average speed up of TRUM is almost 80. TRUM achieves the best speed ups in datasets *pumsb_star*, *retail* and *T10I4D100K*, where TRUM outperforms GC-growth by 2 orders of magnitude.

In addition, we observe that, in batch maintenance, the execution time of discovery algorithms decreases slightly as the update interval increases. This is because, when more transactions are removed, the dataset becomes smaller. Thus it takes less time to discover patterns from it. More importantly, we also observe that the advantage of TRUM over discovery algorithms diminishes as the size of update increases. This is because larger update size logically leads to more dramatic changes to the frequent pattern space and makes the pattern space computationally more expensive to be maintained. In batch maintenance, it is inevitable that when the amount of update increases to a certain extent, the changes induced to the pattern space become so significant that it becomes more efficient to re-discover the pattern space than to maintain and update it.

The next question is: when should one switch from maintaining the frequent pattern space with TRUM to re-discovering the pattern space with discovery algorithms? Experiments were conducted to investigate this and to identify the "switching point".



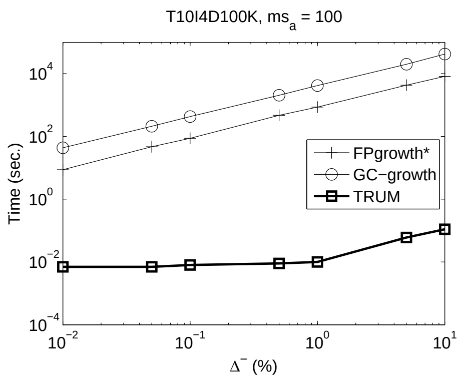


Figure 5.4: Performance comparison of TRUM and the discovery algorithms, FP-growth* and GC-growth, under eager maintenance. Notations: $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$.

Here, the "switching point" refers to the point of update inteval, Δ^-, at which re-discovery algorithms start to outperform TRUM. We discover that the "switching points" vary across different datasets, and, even for a particular dataset, different "switching points" are observed under different minimum support thresholds. As a result, it is impossible to suggest a universal "switching point" for various datasets and minimum support thresholds. However, through our experimental studies, as demonstrated in Figure 5.3, we conclude that TRUM is a more effective option compared to re-discovering the frequent pattern space as far as $\Delta^- \leq 10\%$.

Eager Maintenance

In eager maintenance, decremental transactions in $\mathcal{D}_{dec}$ are removed from the original dataset $\mathcal{D}_{org}$ one by one. Discovery algorithms need to be applied to re-discover the frequent pattern space for each removal. It is obvious that, in eager maintenance, rediscovery the pattern space involves high redundancy and is hence extremely "costly". Maintaining the pattern space with TRUM is a much more effective solution. Some experimental results are presented in Figure 5.4 as examples.

As can be seen in Figure 5.4, in eager maintenance, the execution time of the discovery algorithms grows steeply as the update interval increases. The advantage of TRUM gets more and more significant as more transactions are removed from the original dataset in an "eager" manner.

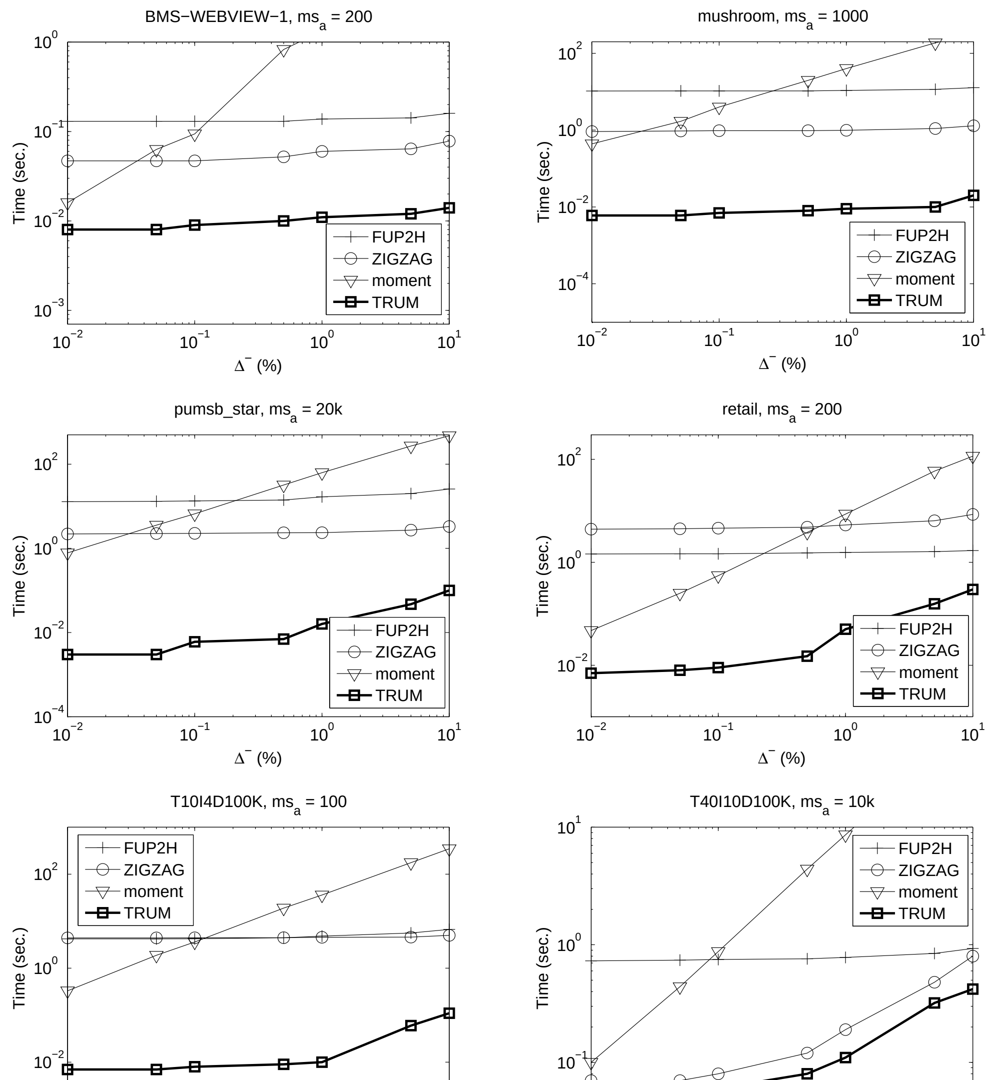


Figure 5.5: Performance comparison of TRUM and the pattern maintenance algorithms: FUP2H, ZIGZAG and moment . Notations: $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$.

5.3.2 Comparison with Maintenance Algorithms

TRUM is also compared with the representative maintenance algorithms FUP2H [15], ZIGZAG [70] and moment [18]. FUP2H, which is the generalized version of FUP [14], maintains all frequent patterns. ZIGZAG and moment are recently proposed maintenance algorithms. ZIGZAG maintains the maximal frequent patterns, and moment maintains the closed patterns. Pattern maintenance algorithms, unlike discovery algorithms, do not suffer from large extra overheads in eager maintenance. Therefore, we compare the efficiency of TRUM and maintenance algorithms only under the setting of batch maintenance.

Figure 5.5 graphically compares the performance of TRUM with the representative maintenance algorithms. TRUM outperforms the other maintenance algorithms significantly. As summarized in Table 5.2, on average, TRUM is almost 6000 times faster than FUP2H, more than 100 times faster than ZIGZAG and almost 3000 times faster moment. For the best cases, TRUM outperforms ZIGZAG by almost 3 orders of magnitude and outperforms both FUP2H and moment more than 4 orders of magnitude.

Dataset	Discovery Algorithms		Discovery Algorithms		
	FPgrowth*	GC-growth	FUP2H	ZIGZAG	moment
BMS-WEBVIEW-1 ($ms_a = 200$)	7	53	14	5.5	192
BMS-WEBVIEW-1 ($ms_a = 100$)	9	69	33	20	227
BMS-WEBVIEW-2 ($ms_a = 200$)	18	32	57	69	248
BMS-WEBVIEW-2 ($ms_a = 100$)	17	32	51	174	175
chess ($ms_a = 2k$)	43	11	870	62	3,520
chess ($ms_a = 1.5k$)	130	13	3,640	28	**10,600**
connect-4 ($ms_a = 40k$)	130	3	**30,000**	140	1,180
connect-4 ($ms_a = 30k$)	24	1.5	4,600	10	182
mushroom ($ms_a = 1k$)	128	30	1,325	122	6,200
mushroom ($ms_a = 500$)	**1,240**	**31**	**17,000**	486	**10,700**
pumsb ($ms_a = 40k$)	8	9.2	1,420	1.5	585
pumsb ($ms_a = 35k$)	77	12	28,000	2	**14,600**
pumsb_star ($ms_a = 20k$)	78	127	2,084	348	3,071
pumsb_star ($ms_a = 15k$)	**255**	**72**	**29,000**	**375**	**7,950**
retail ($ms_a = 200$)	60	186	100	314	184
retail ($ms_a = 100$)	58	306	178	**818**	208
T10I4D100K ($ms_a = 500$)	64	113	87	90	1,288
T10I4D100K ($ms_a = 100$)	**80**	**400**	**410**	**413**	**1,800**
T40I10D100K ($ms_a = 10k$)	9	7	9	1.5	69
T40I10D100K ($ms_a = 5k$)	5	3	6	1.5	6
Average	119	79.6	5,944	174	2,848

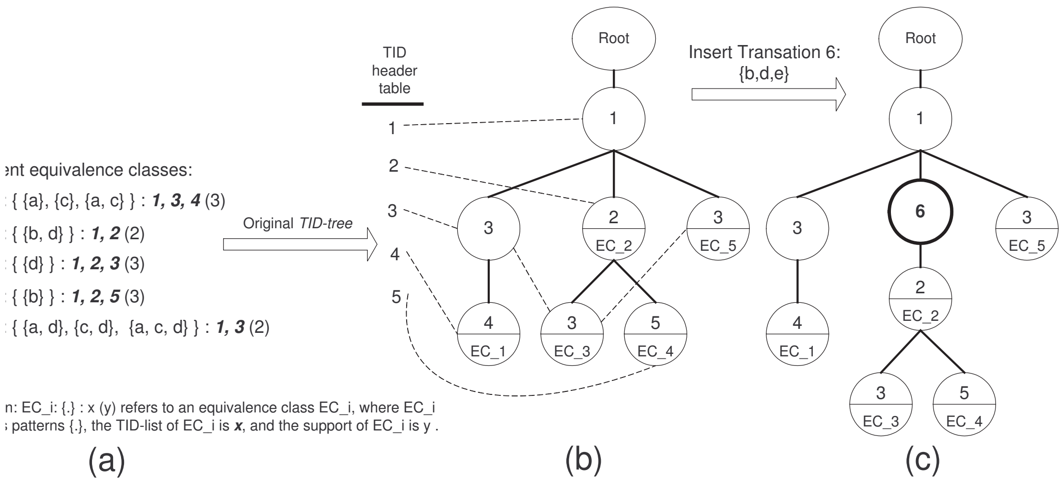
Table 5.2: Average speed up of TRUM over benchmark datasets for batch maintenance (when $\Delta^- \leq 10\%$).

5.4 Generalization & Extension

5.4.1 Extension for Percentage Support Threshold

TRUM is developed under the setting, where minimum support threshold, ms_a, is defined in terms of absolute counts. However, in some applications, the minimum support threshold may be defined in terms of percentages, $ms_{\%}$. When the minimum support threshold is defined in terms of percentage, the absolute support threshold, ms_a, will actually drop after decremental updates. Thus new frequent patterns and equivalence classes may emerge. TRUM cannot be directly applied to this case, and some extension to the algorithm is required.

We here introduce one possible way of extension. Suppose the percentage threshold $ms_{\%} = x\%$. In the extended version of TRUM, the *TID-tree* not only stores the frequent equivalence classes, and the *TID-tree* also includes the "buffer equivalence classes", whose supports are above $(x - \triangle)\%$. In this way, we actually build a "buffer" in the *TID-tree*, which contains a group of infrequent equivalence classes that are likely to become frequent after some existing transactions are removed. This means that, so long as the total number of deletions does not exceed $|\mathcal{D}_{org}| \times \triangle\%$, all the "new" equivalence classes that may emerge are already kept in the "buffer" of the extended *TID-tree*. With the extended *TID-tree*, TRUM can now be employed for multiple rounds of decremental maintenance, as long as the accumulated amount of deletion is less than $|\mathcal{D}_{org}| \times \triangle\%$. The size of the $\triangle\%$ buffer can be adjusted based on specific application requirements.



ent equivalence classes:
{ {a}, {c}, {a, c} } : 1, 3, 4 (3)
{ {b, d} } : 1, 2 (2)
{ {d} } : 1, 2, 3 (3)
{ {b} } : 1, 2, 5 (3)
{ {a, d}, {c, d}, {a, c, d} } : 1, 3 (2)
n: EC_i: {.} : x (y) refers to an equivalence class EC_i, where EC_i
patterns {.}, the TID-list of EC_i is x, and the support of EC_i is y .
Original TID-tree
TID header table
Root
Insert Transation 6: {b,d,e}
EC_1
EC_2
EC_3
EC_4
EC_5
(a)
(b)
(c)

Figure 5.6: *TID-tree* for incremental updates. (a) The existing frequent equivalence classes; (b) the original *TID-tree*; and (c) the updated *TID-tree* after inserting transaction 6: $\{b, d, e\}$.

The above extension of TRUM is not a perfect solution, and it has its limitation. As the amount of deletion gets close to the limit $|\mathcal{D}_{org}| \times \triangle\%$, we have to re-discover the frequent pattern space and rebuild the buffer. To solve the problem completely, extra data structure, e.g. *GE-tree* (which will be introduced in next chapter), is required.

5.4.2 Generalization to Incremental Updates

TID-tree is demonstrated to be an effective data structure for decremental maintenance of frequent patterns. Under decremental updates, *TID-tree* can be efficiently updated by removing the nodes corresponds to the deleted transactions. It is very tempting to generalize *TID-tree* to address also incremental updates. However, we found that it is computationally expensive to update *TID-tree* under incremental updates.

Let us illustrate this with the example shown in Figure 5.1. Suppose a new transaction $\{b, d, e\}$ is inserted to the dataset, and, as shown in Figure 5.6, we name the new transaction with a unique ID 6. To insert transaction 6 into the *TID-tree*, we first need to scan through all existing equivalence classes and compare them with the new transaction. This step is to determine which equivalence classes are affected by the incremental update. In this case, equivalence classes EC_2, EC_3 and EC_4 are affected. We then compare equivalence classes EC_2, EC_3 and EC_4 again to determine where the new node/nodes of transaction 6 should be inserted into the tree. In this example, it requires 8 comparisons of equivalence classes to insert a single transaction to the *TID-tree*. We observe that the insertion of new transactions into *TID-tree* is a much more complicated process than the removal one. Therefore, we believe *TID-tree* is not an effective data structure for incremental maintenance.

5.5 Summary

In this chapter, we have proposed a novel algorithm — TRUM — to maintain the frequent pattern space for decremental updates. TRUM is inspired by the space evolution analysis discussed in Chapter 4. TRUM addresses the decremental maintenance problem in a divide-and-conquer manner by decomposing the frequent pattern space into equivalence classes. We have also developed a new data structure, *TID-tree*, to facilitate the computational tasks involved in TRUM. With *TID-tree*, TRUM effectively maintains the pattern space by updating only the equivalence classes that are affected by the decremental update.

Extensive experiments are conducted to evaluate the effectiveness of TRUM. According to the empirical results, TRUM significantly outperforms the representation frequent pattern discovery and maintenance algorithms.

In addition, we demonstrated that: although TRUM is developed for the case, where the minimum support threshold is defined in terms of absolute count, TRUM can be extended to address also the case, where the threshold is defined in terms of percentage. We also discovered that the idea of *TID-tree* cannot be effectively generalized to incremental updates.

Chapter 6

Pattern Space Maintainer (PSM): A Complete Maintainer

This chapter proposes a complete maintainer for the space of frequent patterns. The proposed maintainer is called the "Pattern Space Maintainer", in short PSM. PSM addresses the incremental maintenance, decremental maintenance and support threshold adjustment maintenance of frequent pattern space. Since PSM addresses all 3 different update scenarios, it is composed of 3 maintenance components: the incremental maintenance component, PSM+, the decremental maintenance component, PSM- and the threshold adjustment maintenance component, PSM_Δ. All three components are proposed based on the pattern space evolution analysis in Chapter 4 Based on the space evolution analysis, we summarize the major computational tasks involved in the maintenance components. To effectively perform the maintenance computational tasks, we devlope a new maintenance data structure — "Generator-Enumeration Tree" (*GE-tree*). The three maintenance components are all constructed based on

GE-tree, and thus they can be easily integrated together. However, for ease of presentation, we introduce the three maintenance components separately in this chapter.

6.1 Incremental Maintenance

This section introduces the incremental maintenance component of PSM — PSM+. In the incremental update, a set of new transactions $\mathcal{D}_{inc}$ are inserted into the original dataset $\mathcal{D}_{org}$, and thus the updated dataset $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \mathcal{D}_{inc}$. Given a support threshold, the task of incremental maintenance is to obtain the updated pattern space by maintaining the original pattern space.

PSM+ is proposed based on the pattern space evolution analysis. Thus we first recap how the space of frequent patterns evolves under incremental updates. Based on the evolution analysis, we further summarize the major computational tasks of incremental maintenance. We then define the concept of "Generator-Enumeration Tree" (*GE-tree*), and we also illustrate how *GE-tree* helps PSM+ to complete the computational tasks efficiently. Lastly, the correctness and effectiveness of PSM+ are demonstrated through both complexity analysis and experimental studies.

6.1.1 Evolution of Pattern Space

Under incremental updates, one existing equivalence class may evolve in three different ways: it may remain unchanged without any change in support; or it may remain unchanged but with an increased support; or it may split into multiple classes. The exact conditions for each evolution scenarios are described in Theorem 4.16.

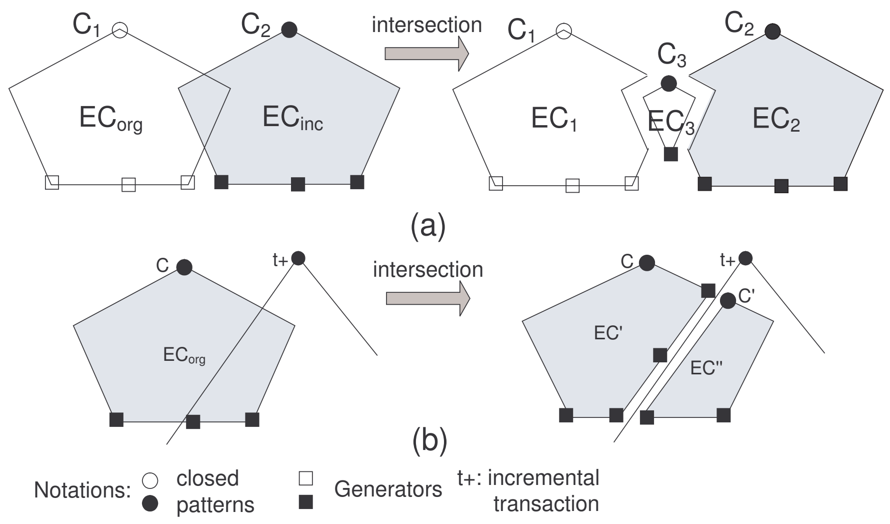


Figure 6.1: (a) Split up of the existing equivalence class EC_{org} by intersecting with EC_{inc}, an equivalence class in the incremental dataset, and (b) split up of EC_{org} by intersecting with a single incremental transaction $t+$.

Among all evolution scenarios, the splitting scenarios are the most complicated ones. As demonstrated in Figure 6.1 (a), the splitting up of an existing equivalence class can be achieved by intersecting the existing class with the classes in the incremental dataset. However, this intersection process can be pretty "messy". Figure 6.1 (a) shows an example. After the intersection, equivalence classes EC_{org} and EC_{inc} split into three parts, EC_1, EC_2 and EC_3. The messiness comes from the fact that EC_1 and EC_2 are not convex. Although we have demonstrated in theory that EC_1 and EC_2 will further split into multiple convex equivalence classes, it is computational challenging in practice to complete the spilt up. Moreover, since EC_1 and EC_2 are not convex, we can no longer concisely represent them using their borders, which makes it extremely difficult to store and identify them for further split ups.

To simplify the split up situation, we propose to address the update problem in a divide-and-conquer manner. In particular, we propose to update the frequent pattern space iteratively by considering only one incremental transaction at a time. As illustrated in Figure 6.1 (b), if we consider only one incremental transaction $t+$, the existing equivalence class EC_{org} splits into only two parts: EC' and EC''. More importantly, both EC' and EC'' are convex equivalence classes. This observation is formally summarized in Proposition 6.1. In Proposition 6.1, we use $Close(X)$ and $Keys(X)$ to indicate the closed pattern and generators (key patterns) of an equivalence class X.

Proposition 6.1. *Let $[P]_{\mathcal{D}_{org}}$ be an existing equivalence class in $\mathcal{D}_{org}$. Also let $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup t_+$, where t_+ is an incremental transaction. Then, in $\mathcal{D}_{upd+}$, $[P]_{\mathcal{D}_{org}}$ splits into two equivalence classes, iff $\exists Q \in [P]_{\mathcal{D}_{org}}$ such that $Q \not\subseteq t_+$ and $\exists Q' \in [P]_{\mathcal{D}_{org}}$ such that $Q' \subseteq t_+$. In particular, $[P]_{\mathcal{D}_{org}}$ splits into class $[Q]_{\mathcal{D}_{upd+}}$ (corresponds to EC' in Figure 6.1) and class $[Q']_{\mathcal{D}_{upd+}}$ (corresponds to EC'' in Figure 6.1).*

Moreover, $[Q]_{\mathcal{D}_{upd+}} = \{X | X \in [P]_{\mathcal{D}_{org}} \wedge X \not\subseteq t_+\}$, $sup(Q, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org})$, $Close([Q]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}})$ and $Keys([Q]_{\mathcal{D}_{upd+}}) = min\{\{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \not\subseteq t_+\} \cup \{K' \cup \{x_i\}, i = 1, 2, \cdots | K' \in Keys([P]_{\mathcal{D}_{org}}) \wedge K' \subseteq t_+, x_i \in Close([P]_{\mathcal{D}_{org}}) \wedge x_i \notin t_+\}\}$. On the other hand, $[Q']_{\mathcal{D}_{upd+}} = \{Y | Y \in [P]_{\mathcal{D}_{org}} \wedge Y \subseteq t_+\}$, $sup(Q', \mathcal{D}_{upd+}) = sup(Q', \mathcal{D}_{org}) + 1$, $Close([Q]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}}) \cap t_+$ and $Keys([Q']_{\mathcal{D}_{upd+}}) = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$.

Proof. We first prove the left-to-right direction of the first part of the proposition. Suppose $[P]_{\mathcal{D}_{org}}$ splits into $[Q]_{\mathcal{D}_{upd+}}$ and $[Q']_{\mathcal{D}_{upd+}}$ in $\mathcal{D}_{upd+}$. This means $[Q]_{\mathcal{D}_{upd+}} \cup [Q']_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{upd+}} \cap [Q']_{\mathcal{D}_{upd+}} = \emptyset$. It is obvious that Q & $Q' \in [P]_{\mathcal{D}_{org}}$. Thus, according to the definition of equivalence class (Definition 4.3), $f(Q, \mathcal{D}_{org}) =$

$f(Q', \mathcal{D}_{org})$. Also since $[Q]_{\mathcal{D}_{upd+}} \cap [Q']_{\mathcal{D}_{upd+}} = \emptyset$, $f(Q, \mathcal{D}_{upd+}) \neq f(Q', \mathcal{D}_{upd+})$. Let us assume without loss of generality that $f(Q, \mathcal{D}_{upd+}) \subset f(Q', \mathcal{D}_{upd+})$. Since we only insert one incremental transaction t_+ to $\mathcal{D}_{org}$, the only possibility is that $f(Q, \mathcal{D}_{upd+}) = f(Q, \mathcal{D}_{org})$ and $f(Q', \mathcal{D}_{upd+}) = f(Q', \mathcal{D}_{org}) \cup \{t_+\} \supset f(Q, \mathcal{D}_{upd+})$. This implies that $Q \not\subseteq t_+$ and $Q' \subseteq t_+$. The left-to-right direction is proven.

For the right-to-left direction, suppose $\exists Q \in [P]_{\mathcal{D}_{org}}$ that $Q \not\subseteq t_+$ and $\exists Q' \in [P]_{\mathcal{D}_{org}}$ that $Q' \subseteq t_+$. This implies $f(Q, \mathcal{D}_{upd+}) = f(Q, \mathcal{D}_{org})$ and $f(Q', \mathcal{D}_{upd+}) = f(Q', \mathcal{D}_{org}) \cup \{t_+\}$. Moreover, according to Proposition 4.10, equivalence classes will only shrink after incremental updates. Thus $[Q]_{\mathcal{D}_{upd+}} \subset [P]_{\mathcal{D}_{org}}$ and $[Q']_{\mathcal{D}_{upd+}} \subset [P]_{\mathcal{D}_{org}}$. Combining this with the definition of equivalence class, we then have $[Q]_{\mathcal{D}_{upd+}} = \{X | X \in [P]_{\mathcal{D}_{org}} \wedge f(X, \mathcal{D}_{upd+}) = f(Q, \mathcal{D}_{upd+})\} = \{X | X \in [P]_{\mathcal{D}_{org}} \wedge X \not\subseteq t_+\}$ and $[Q']_{\mathcal{D}_{upd+}} = \{Y | Y \in [P]_{\mathcal{D}_{org}} \wedge f(Y, \mathcal{D}_{upd+}) = f(Q', \mathcal{D}_{upd+})\} = \{Y | Y \in [P]_{\mathcal{D}_{org}} \wedge Y \subseteq t_+\}$. It is obvious that $[Q]_{\mathcal{D}_{upd+}} \cap [Q']_{\mathcal{D}_{upd+}} = \emptyset$ and $[Q]_{\mathcal{D}_{upd+}} \cup [Q']_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$. The right-to-left direction is proven.

We now prove the second part of the proposition. It has already been proven in the last paragraph that $[Q]_{\mathcal{D}_{upd+}} = \{X | X \in [P]_{\mathcal{D}_{org}} \wedge X \not\subseteq t_+\}$ and $[Q']_{\mathcal{D}_{upd+}} = \{Y | Y \in [P]_{\mathcal{D}_{org}} \wedge Y \subseteq t_+\}$.

For $[Q]_{\mathcal{D}_{upd+}}$, since $Q \not\subseteq t_+$, $f(Q, \mathcal{D}_{upd+}) = f(Q, \mathcal{D}_{org})$. Thus $sup(Q, \mathcal{D}_{upd+}) = sup(Q, \mathcal{D}_{org})$. Next, we prove that $Close([Q]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}})$. Let $C = Close([P]_{\mathcal{D}_{org}})$. Then we have $C \supseteq Q$ and, thus, $C \not\subseteq t_+$. This implies that $C \in [Q]_{\mathcal{D}_{upd+}}$. Also since C is the closed pattern in $[P]_{\mathcal{D}_{org}}$, C is the most specific pattern in $[P]_{\mathcal{D}_{org}}$ (Definition 4.6). This further implies that C is also the most specific pattern in $[Q]_{\mathcal{D}_{upd+}}$ for $[Q]_{\mathcal{D}_{upd+}} \subset [P]_{\mathcal{D}_{org}}$. Therefore, C is also the closed pattern of

$[Q]_{\mathcal{D}_{upd+}}$, and $Close([Q]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}})$. For the formula of $Keys([Q]_{\mathcal{D}_{upd+}})$, it basically refers to the set of minimum (most general) patterns in $[Q]_{\mathcal{D}_{upd+}}$. This follows directly from Definition 4.6.

For $[Q']_{\mathcal{D}_{upd+}}$, since $Q' \subseteq t_+$, $f(Q', \mathcal{D}_{upd+}) = f(Q', \mathcal{D}_{org}) \cup \{t_+\}$. Thus $sup(Q', \mathcal{D}_{upd+}) = |f(Q', \mathcal{D}_{upd+})| = sup(Q', \mathcal{D}_{org}) + 1$.

Next, we prove $Close([Q']_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}}) \cap t_+$. Let $C = Close([P]_{\mathcal{D}_{org}}) \cap t_+$. It is obvious that (1) $C \subseteq Close([P]_{\mathcal{D}_{org}})$ and (2) $C \subseteq t_+$. According to the definition of Q', $Q' \subseteq t_+$ and $Q' \subseteq Close([P]_{\mathcal{D}_{org}})$ (for $Q' \in [P]_{\mathcal{D}_{org}}$), thus $Q' \subseteq Close([P]_{\mathcal{D}_{org}}) \cap t_+$, meaning (3) $Q' \subseteq C$. According to the definition of convex space, point (1) & (3) imply that $C \in [P]_{\mathcal{D}_{org}}$. Combining the facts that $C \in [P]_{\mathcal{D}_{org}}$ and $C \subseteq t_+$, we have $C \in [Q']_{\mathcal{D}_{upd+}}$. We then assume that there exists C' such that $C' \supset C$ and $C' \in [Q']_{\mathcal{D}_{upd+}}$. $C' \in [Q']_{\mathcal{D}_{upd+}}$ implies that $C' \in [P]_{\mathcal{D}_{org}}$ and $C' \subseteq t_+$. $C' \in [P]_{\mathcal{D}_{org}}$ further implies that $C' \subseteq Close([P]_{\mathcal{D}_{org}})$. Then we have $C' \subseteq Close([P]_{\mathcal{D}_{org}})$ and $C' \subseteq t_+$, and thus $C' \subseteq C$ (for $C = Close([P]_{\mathcal{D}_{org}}) \cap t_+$). This contradicts with the initial assumption. Therefore, $C \in [Q']_{\mathcal{D}_{upd+}}$ and $\nexists C'$ such that $C' \supset C$ and $C' \in [P]_{\mathcal{D}_{upd+}}$. According to Definition 4.6, C is the closed pattern of $[Q']_{\mathcal{D}_{upd+}}$.

Then we prove $Keys([Q']_{\mathcal{D}_{upd+}}) = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$. First, let $\mathcal{K} = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$ and let pattern X be any pattern such that $X \in \mathcal{K}$. $X \in \mathcal{K}$ implies that $X \in [P]_{\mathcal{D}_{org}}$ and $X \subseteq t_+$. This means $X \in [Q']_{\mathcal{D}_{upd+}}$. $X \in \mathcal{K}$ also means $X \in Keys([P]_{\mathcal{D}_{org}})$, i.e. X is one of the most "general" patterns in $[P]_{\mathcal{D}_{org}}$ (Definition 4.6). Moreover, $[P]_{\mathcal{D}_{upd+}} \subset [P]_{\mathcal{D}_{org}}$. Therefore, X must also be one of the most "general" patterns in $[Q']_{\mathcal{D}_{upd+}}$. This means that $X \in Keys([Q']_{\mathcal{D}_{upd+}})$ for every $X \in \mathcal{K}$. Thus we have (A) $\mathcal{K} \subseteq Keys([P]_{\mathcal{D}_{upd+}})$. Second, we assume that there exists a pattern Y such that $Y \in Keys([Q']_{\mathcal{D}_{upd+}})$ but $Y \notin \mathcal{K}$. $Y \in Keys([Q']_{\mathcal{D}_{upd+}})$ means $Y \in [Q']_{\mathcal{D}_{upd+}}$. According to the definition of $[Q']_{\mathcal{D}_{upd+}}$, we know $Y \in [P]_{\mathcal{D}_{org}}$ and $Y \subseteq t_+$. $Y \subseteq t_+$ and $Y \notin \mathcal{K}$ imply that $Y \notin Keys([P]_{\mathcal{D}_{org}})$. This means there exists pattern $K' \subset Y$ such that $K' \in [P]_{\mathcal{D}_{org}}$ (Definition 4.6). Since $K' \subset Y$ and $Y \subseteq t_+$, $K' \subset t_+$, which implies $K' \in [Q']_{\mathcal{D}_{upd+}}$. Thus, according to Definition 4.6, $Y \notin Keys([P]_{\mathcal{D}_{upd+}})$. This contradicts with the initial assumption. Thus there does not exists pattern Y such that $Y \in Keys([Q']_{\mathcal{D}_{upd+}})$ but $Y \notin \mathcal{K}$. Therefore, we have (B) $\mathcal{K} \supseteq Keys([P]_{\mathcal{D}_{upd+}})$. Combining results (A) and (B), we have $Keys([P]_{\mathcal{D}_{upd+}}) = \mathcal{K} = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$. The proposition is proven. □

Furthermore, we can also simplify Theorem 4.16 by considering only one incremental transaction at a time.

Theorem 6.2. *Let $\mathcal{D}_{org}$ be the original dataset, t_+ be the incremental transaction, $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \{t_+\}$ and $ms_{\%}$ be the support threshold. For every frequent equivalence class $[P]_{\mathcal{D}_{upd+}}$ in $\mathcal{F}(ms_{\%}, \mathcal{D}_{upd+})$, exactly one of the 5 scenarios below holds:*

1. *$P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, $P \not\subseteq t_+$ and $Q \not\subseteq t_+$ for all $Q \in [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where the equivalence class remains totally unchanged. In this case, $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$ and $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org})$.*

2. *$P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, $P \subseteq t_+$ and $Q \subseteq t_+$ for all $Q \in [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where the equivalence class has remained unchanged but with increased support. In this case, $[P]_{\mathcal{D}_{upd+}} = [P]_{\mathcal{D}_{org}}$ and $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + 1$.*

3. *$P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, $P \subseteq t_+$ and $Q \not\subseteq t_+$ for some $Q \in [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where the equivalence class splits. In this case, $[P]_{\mathcal{D}_{org}}$ splits*

into two new equivalence classes, and $[P]_{\mathcal{D}_{upd+}}$ *is one of them.* $[P]_{\mathcal{D}_{upd+}} = \{Q | Q \in [P]_{\mathcal{D}_{org}} \wedge Q \subseteq t_+\}$*,* $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + 1$*,* $Close([P]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}}) \cap t_+$ *and* $Keys([P]_{\mathcal{D}_{upd+}}) = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$*.*

4. $P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$*,* $P \nsubseteq t_+$ *and* $Q \subseteq t_+$ *for some* $Q \in [P]_{\mathcal{D}_{org}}$*, also corresponding to the scenario where the equivalence class splits. This scenario is complement to Scenario 3.* $[P]_{\mathcal{D}_{org}}$ *splits into two new equivalence classes,* $[P]_{\mathcal{D}_{upd+}}$ *is one of them, and the other one has been described in Scenario 3. In this case,* $[P]_{\mathcal{D}_{upd+}} = \{Q | Q \in [P]_{\mathcal{D}_{org}} \wedge Q \nsubseteq t_+\}$*,* $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org})$*,* $Close([P]_{\mathcal{D}_{upd+}}) = Close([P]_{\mathcal{D}_{org}})$ *and* $Keys([P]_{\mathcal{D}_{upd+}}) = min\Big\{\{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \nsubseteq t_+\} \cup \{K' \cup \{x_i\}, i = 1, 2, \cdots | K' \in Keys([P]_{\mathcal{D}_{org}}) \wedge K' \subseteq t_+, x_i \in Close([P]_{\mathcal{D}_{org}}) \wedge x_i \notin t_+\}\Big\}$*.*

5. $P \notin \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$*,* $P \subseteq t_+$ *and* $Sup(P, \mathcal{D}_{upd+}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil$*, corresponding to the scenario where a new frequent equivalence class has emerged. In this case,* $[P]_{\mathcal{D}_{upd+}} = \{Q | Q \in [P]_{\mathcal{D}_{org}} \wedge Q \subseteq t_+\}$ *and* $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + 1$*.*

Proof. Scenarios 1 and 5 are obvious. Scenario 3 follows Proposition 6.1. $[P]_{\mathcal{D}_{upd+}}$ in Scenario 3 is equivalent to $[Q']_{\mathcal{D}_{upd+}}$ in Proposition 6.1. Scenario 4 is complementary to Scenario 3, and it also follows Proposition 6.1. $[P]_{\mathcal{D}_{upd+}}$ in Scenario 4 is equivalent to $[Q]_{\mathcal{D}_{upd+}}$ in Proposition 6.1.

To prove Scenario 2, suppose (i) $P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, (ii) $P \subseteq t_+$ and (iii) $Q \subseteq t_+$ for all $Q \in [P]_{\mathcal{D}_{org}}$. Point (ii) implies that $f(P, \mathcal{D}_{upd+}) = f(P, \mathcal{D}_{org}) \cup \{t_+\}$, and point (iii) implies that, for all $Q \in [P]_{\mathcal{D}_{org}}$, $f(Q, \mathcal{D}_{upd+}) = f(Q, \mathcal{D}_{org}) \cup \{t_+\}$. According to the definition of equivalence class (Definition 4.3), $f(P, \mathcal{D}_{org}) = f(Q, \mathcal{D}_{org})$. Thus $f(P, \mathcal{D}_{upd+}) = f(P, \mathcal{D}_{org}) \cup \{t_+\} = f(Q, \mathcal{D}_{org}) \cup \{t_+\} = f(Q, \mathcal{D}_{upd+})$. This means that, for all $Q \in [P]_{\mathcal{D}_{org}}$, $Q \in [P]_{upd+}$. Therefore, the equivalence $[P]_{\mathcal{D}_{org}}$ remains the same after the update, but $sup(P, \mathcal{D}_{upd+}) = |f(P, \mathcal{D}_{upd+})| = sup(P, \mathcal{D}_{org}) + 1$.

Finally, we prove that the theorem is complete. For patterns $P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, it is obvious that Scenario 1 to 4 enumerated all possible cases. For pattern $P \notin \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, Scenario 5 corresponds to the case where $P \subseteq t_+$ and $Sup(P, \mathcal{D}_{upd+}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil$. The cases where $P \nsubseteq t_+$ or $Sup(P, \mathcal{D}_{upd+}) < \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil$ are not enumerated, because, in these cases, it is clear that $P \notin \mathcal{F}(ms_{\%}, \mathcal{D}_{upd+})$. As a result, we can conclude that this theorem is sound and complete. $\square$

Theorem 6.2 summarizes how the frequent pattern space evolves when one incremental transaction is inserted. More importantly, the theorem describes how the borders of existing frequent equivalence classes can be updated after the insertion of the incremental transaction.

In addition, recall that if the support threshold is defined in terms of percentage, $ms_{\%}$, an incremental update may cause the absolute support threshold ms_a to increase. This increase in ms_a may invalidate existing frequent equivalence classes.

Combining all the above observations, we summarize that the incremental maintenance of the frequent pattern space involves four major **computational tasks**: (1) update the support of existing frequent equivalence classes; (2) split up equivalence classes that satisfy Scenario 3 and 4 of Theorem 6.2; (3) discover newly emerged frequent equivalence classes; and (4) remove existing frequent equivalence classes that are no longer frequent. Task (4) is not required if the support threshold is defined in terms of absolute count. For the case where percentage support threshold is used, Task (4) can be accomplished by filtering out the infrequent equivalence classes when outputting them. This filtering step is very straightforward, and thus we will not elaborate its details. We here focus on the first three tasks, and we name them respectively as the **Support Update** task, **Class Splitting** task and **New Class Discovery** task. To efficiently complete these three tasks, a new data structure, *Generator-Enumeration Tree* (*GE-tree*), is developed.

6.1.2 Maintenance Data Structure: Generator-Enumeration Tree (*GE-tree*)

The *Generator-Enumeration Tree* (*GE-tree*) is a data structure inspired by the idea of the *Set-Enumeration Tree* (*SE-tree*). Thus we first recap the concept of *SE-tree*. We then introduce the characteristics of *GE-tree*, and we further demonstrate how the *GE-tree* can help to efficiently complete the computational tasks of incremental maintenance.

6.1.2.1 Set-Enumeration Tree

Set-Enumeration Tree (*SE-tree*), as shown in Figure 6.2, is a conceptual data structure that guides the systematic enumeration of patterns.

Let the set $I = \{i_1, ..., i_m\}$ of items be ordered according to an arbitrary ordering $<_0$ so that $i_1 <_0 i_2 <_0 \cdots <_0 i_m$. For itemsets X, $Y \subseteq I$, we write $X <_0 Y$ iff X

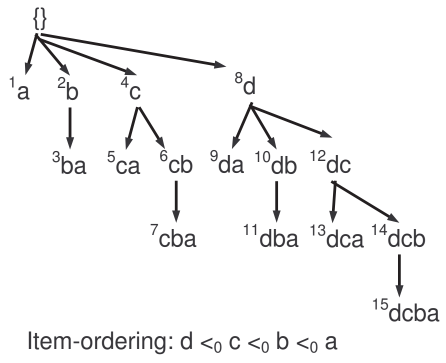


Figure 6.2: The Set-Enumeration Tree with item order: $d <_0 c <_0 b <_0 a$. The number on the left top corner of each node indicates the order at which the node is visited.

is lexicographically "before" Y according to the order $<_0$. E.g. $\{i_1\} <_0 \{i_1, i_2\} <_0 \{i_1, i_3\}$. We say an itemset X is a "prefix" of an itemset Y iff $X \subseteq Y$ and $X <_0 Y$. We write $last(X)$ for the item $\alpha \in X$, if the items in X are $\alpha_1 <_0 \alpha_2 <_0 \cdots <_0 \alpha$. We say an itemset X is the "precedent" of an itemset Y iff $X = Y - last(Y)$. E.g. pattern $\{d, c\}$ in Figure 6.2 (a) is the precedent of pattern $\{d, c, b\}$.

A *SE-tree* is a conceptual organization on the subsets of I so that $\{\}$ is its root node; for each node X such that Y_1, ..., Y_k are all its children from left to right, then $Y_k <_0 \cdots <_0 Y_1$; for each node X in the set-enumeration tree such that X_1, ..., X_k are siblings to its left, we make $X \cup X_1$, ..., $X \cup X_k$ the children of X; $|X \cup X_i| = |X| + 1 = |X_i| + 1$; and $|X| = |X_i| = |X \cap X_i| + 1$. We also induce an enumeration ordering on the nodes of the *SE-tree* so that given two nodes X and Y, we say $X <_1 Y$ iff X would be visited before Y when we visit the set-enumeration tree in a left-to-right top-down manner. Since this visit order is a bit unusual, we illustrate it in Figure 6.2. Here, the number besides the node indicates the order at which the node is visited.

The *SE-tree* is an effective structure for pattern enumeration. Its left-to-right top-down enumeration order effectively ensures complete pattern enumeration without redundancy.

6.1.2.2 Generator-Enumeration Tree

The *Generator-Enumeration Tree* (*GE-tree*) is developed from the *SE-tree*. As shown in Figure 6.3 (a), *GE-tree* is constructed in a similar way as *SE-tree*, and *GE-tree* also follows the left-to-right top-down enumeration order to ensure complete and efficient pattern enumeration.

New features have been introduced to the *GE-tree* to facilitate incremental maintenance of frequent patterns. In the literature, *SE-tree* has been used to enumerate frequent patterns [73], closed patterns [72] and maximal patterns [35]. However, *GE-tree*, as the name suggested, is employed here to enumerate frequent generators. Moreover, unlike *SE-tree*, in which the items are arranged according to some arbitrary order, items in *GE-tree* is arranged based on the support of the items. This means items $i_1 <_0 i_2$ if $sup(\{i_1\}, \mathcal{D}) < sup(\{i_2\}, \mathcal{D})$. This item ordering effectively minimizes the size of the *GE-tree*. Also, different from *SE-tree*, which only acts as a conceptual data structure, *GE-tree* acts as a compact storage structure for frequent generators. As shown in Figure 6.3, each node in *GE-tree* represents a generator, and each frequent generator is linked to its corresponding equivalence class. This feature allows frequent generators and their corresponding equivalence classes to be easily updated in response to updates. The most important feature of *GE-tree* is that: it stores the "negative generator border" in addition to frequent generators. For the *GE-tree*

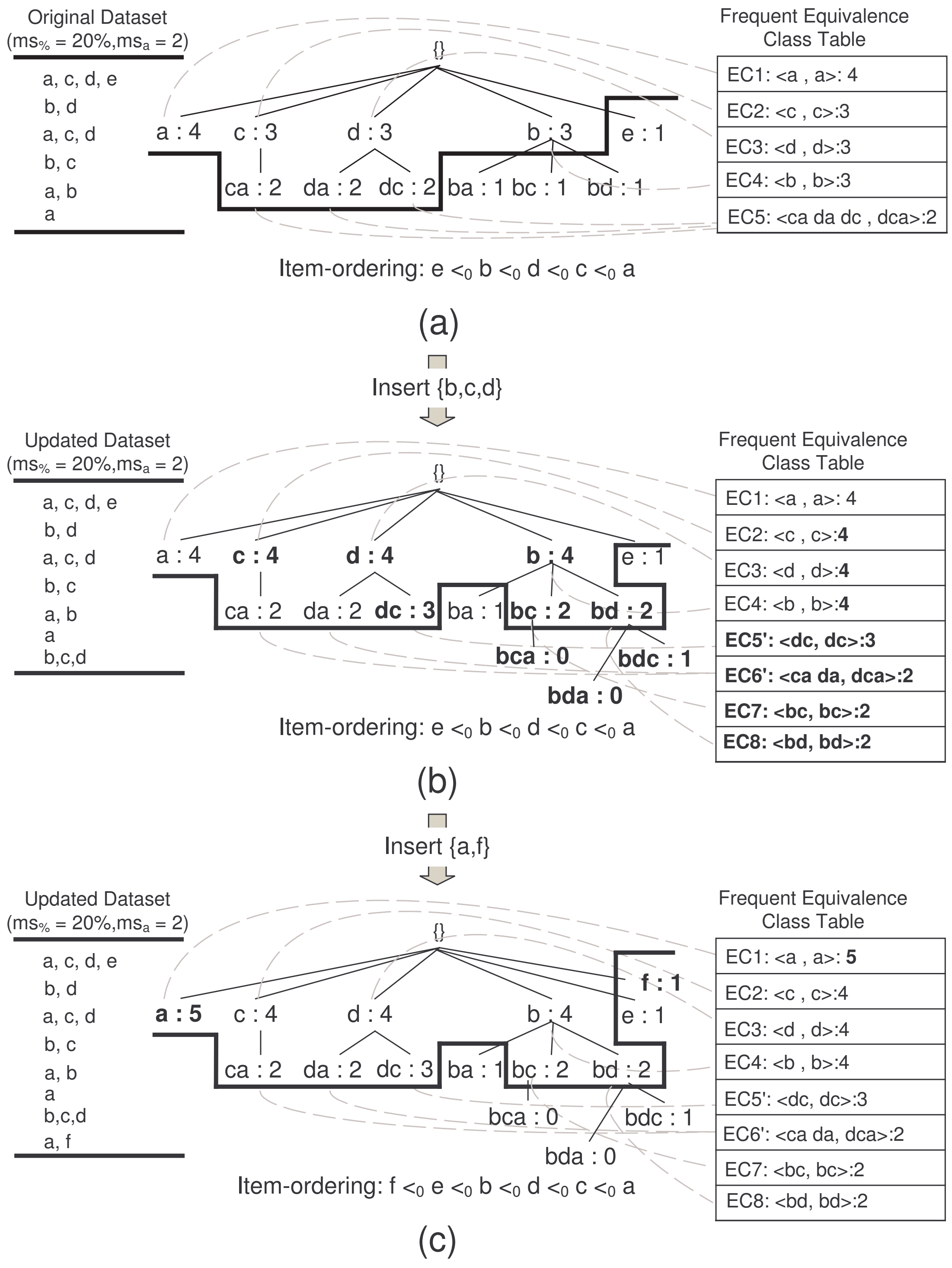


Figure 6.3: (a) The *GE-tree* for the original dataset. (b) The updated *GE-tree* when new transaction $\{b, c, d\}$ is inserted. (c) The updated *GE-tree* when new transaction $\{a, f\}$ is inserted.

in Figure 6.3, the “negative generator border” refers to the collection of generators under the solid line. The “negative generator border” is a newly defined concept for effective enumeration of new frequent generator and equivalence classes.

More details of these new features will be discussed as we demonstrate how *GE-tree* can help to effectively complete the computational tasks of incremental maintenance. Recall that the major computational tasks in the incremental maintenance of the frequent pattern space include the *Support Update* task, *Class Splitting* task and *New Class Discovery* task.

Support Update of existing frequent equivalence classes can be efficiently accomplished with *GE-tree*. The main idea is to update only the frequent equivalence classes that need to be updated. We call these equivalence classes the “affected classes”, and we need a fast way to locate these affected classes.

Since generators are the right bound of equivalence classes, finding frequent generators that need to be updated is equivalent to finding the equivalence classes. *GE-tree* can help us to locate these generators effectively. Suppose a new transaction t_+ is inserted. We will traverse the *GE-tree* in the left-to-right top-down manner. However, we usually do not need to traverse the whole tree. For any generator X in the *GE-tree*, X needs to be updated iff $X \subseteq t_+$. If $X \nsubseteq t_+$, according to Scenario 1 in Theorem 6.2, no update action is needed for X and its corresponding equivalence class. Furthermore, according to the “a priori” property of generators (Fact 4.7), all the children of X can be skipped for the traversal. For example, in Figure 6.3 (c), when transaction $\{a, f\}$ is inserted, only node $\{a\}$ needs to be updated and all the other nodes are skipped.

Class Splitting task can also be completed efficiently with the help of *GE-tree*. The key here is to effectively locate existing frequent equivalence classes that need to be split. Extended from Proposition 6.1, we have the following corollary.

Corollary 6.3. *Suppose a new transaction t_+ is inserted into the original dataset $\mathcal{D}_{org}$. An existing frequent equivalence class $[P]_{\mathcal{D}_{org}}$ splits into two iff $P \subseteq t_+$ but $Close([P]_{\mathcal{D}_{org}}) \nsubseteq t_+$, where $Close([P]_{\mathcal{D}_{org}})$ is the closed pattern of $[P]_{\mathcal{D}_{org}}$.*

Therefore, for an affected class X that has been identified in the support update step, X splits into two iff $Close(X) \nsubseteq t_+$. In Figure 6.3, equivalence class $EC5$ splits into two, $EC5'$ & $EC6'$, after the insertion of $\{b, c, d\}$. This is because pattern $\{c, d\}(\in EC5) \subset \{b, c, d\}$ but $Close(EC5) = \{a, c, d\} \nsubseteq \{b, c, d\}$.

New Class Discovery task is the most challenging computational task involved in the incremental maintenance of the frequent pattern space. This is because, unlike the existing frequent equivalence classes, we have little information about the newly emerged frequent equivalence classes. To address this challenge, a new concept — the "negative generator border" is introduced.

6.1.2.3 Negative Generator Border

The "negative generator border" is defined based on the idea of "negative border". The notion of negative border is first introduced in [51]. The negative border of frequent patterns refers to the set of minimal infrequent patterns. On the other hand, the negative generator border, as formally defined in Definition 6.4, refers to the set of infrequent generators that have frequent precedents in the *GE-tree*. In Figure 6.3, the generators immediately under the solid line are “negative border generators”, and the collection of all these generators forms the “negative generator border”.

Definition 6.4 (Negative Generator Border)**.** *Given a dataset $\mathcal{D}$, support threshold $ms_{\%}$ and the GE-tree, a pattern P is a “negative border generator” iff (1) P is a generator, (2) P is infrequent, (3) the precedent of P in the GE-tree is frequent. The set of all negative border generators is called the “negative generator border”.*

As can be seen in Figure 6.3, the negative generator border records the nodes, where the previous enumeration stops. It thus serves as a convenient starting point for further enumeration of newly emerged frequent generators. This allows us to utilize previously obtained information to avoid redundant generation of existing generator and enumeration of unnecessary candidates.

When new transactions are inserted, the negative generator border is updated along with the frequent generators. Take Figure 6.3 (b) as an example. After the insertion of $\{b, c, d\}$, two negative border generators $\{b, c\}$ and $\{b, d\}$ become frequent. As a result, these two generators will be promoted as frequent generators, and their corresponding equivalence classes $EC7$ and $EC8$ will also be included into the frequent pattern space. Moreover, these two newly emerged frequent generators now act as starting pointing for further enumeration of generators. Following the *SE-tree* enumeration manner, the children of $\{b, c\}$ and $\{b, d\}$ are enumerated by combining $\{b, c\}$ and $\{b, d\}$ with their left hand side siblings, as demonstrated in Figure 6.3 (b). We discover that, after new transactions are added, the negative generator border expands and moves away from the root of *GE-tree*.

Procedure 2 enumNewEC

```
Input: $NG$, a starting point for enumeration; $\mathcal{F}$ the set of frequent equivalence classes;
    $ms_a$ the absolute support threshold and GE-tree.
Output: $\mathcal{F}$ and the updated GE-tree.
Method:
 1: if $NG.support \geq ms_a$ then
 2:   //Newly emerged frequent generator and equivalence class.
 3:   Let $C$ be the corresponding closed pattern of $NG$;
 4:   if $\exists EC \in \mathcal{F}$ such that $EC.close = C$ then
 5:     $NG \to EC.keys$;
        {The corresponding equivalence class already exists.}
 6:   else
 7:     Create new equivalence class $EC'$;
 8:     $EC'.close = C$;
 9:     $NG \to EC'.keys$;
10:     $EC' \to \mathcal{F}$;
11:   end if
      {Enumerate new generators from $NG$}
12:   for all $X$, where $X$ is the left hand side sibling of $NG$ in GE-tree do
13:     $NG' := NG \cup X$;
14:     if $NG'$ is a generator then
15:       enumNewEC($NG'$, $\mathcal{F}$, $ms_a$, GE-tree);
16:     end if
17:   end for
18: else
19:   $NG \to$ GE-tree.ngb; {New negative generator border.}
20: end if
    return $\mathcal{F}$ and GE-tree;
```

The detailed enumeration process is presented in Procedure 2. In Procedure 2 the following notations are used: $NG.support$ denotes the support of generator NG; $EC.close$ refers to the closed pattern of the equivalence class EC; $EC.keys$ refers to the generators of EC and *GE-tree.ngb* refers to the negative generator border of the *GE-tree*.

In summary, *GE-tree* is an effective data structure that not only compactly stores the frequent generators but also guides efficient enumeration of generators. We have demonstrated with examples that the *GE-tree* greatly facilitate the incremental maintenance of the frequent pattern space.

Algorithm 3 PSM+

Input: $\mathcal{D}_{inc}$ the incremental dataset; $|\mathcal{D}_{upd+}|$ the size of the updated dataset; $\mathcal{F}_{org}$ the original frequent pattern space represented using equivalence classes; *GE-tree* and $ms_{\%}$ the support threshold.

Output: $\mathcal{F}_{upd+}$ the update frequent pattern space represented using equivalence classes and the updated *GE-tree*.

Method:

```
1:  $\mathcal{F} := \mathcal{F}_{org}$; {Initialization.}
2:  $ms_a = \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil$;
3:  for all transaction $t$ in $\mathcal{D}_{inc}$ do
4:    for all items $x_i \in t$ that $\{x_i\}$ is not a generator in GE-tree do
5:      $G_{new} := \{x_i\}$, $G_{new}.support := 0$, $G_{new} \rightarrow$ GE-tree;
        {Include new items into GE-tree}
6:    end for
7:    for all generator $G$ in GE-tree that $G \subseteq t$ do
8:      $G.support := G.support + 1$;
9:      if $G$ is an existing frequent generator then
10:       Let $EC$ be the equivalence class of $G$ in $\mathcal{F}$;
11:       if $EC.close \subseteq t$ then
12:         $EC.support = G.support$;{Corresponds to Scenario 2 of Theorem 6.2.}
13:       else
14:         splitEC($\mathcal{F}$, $t$, $G$); {split up equivalence class $EC$. (Procedure 4)}
            {Corresponds to Scenario 3 & 4 of Theorem 6.2.}
15:       end if
16:     else if $G.support \geq ms_a$ then
17:       enumNewEC($G$, $\mathcal{F}$, $ms_a$, GE-tree); {Corresponds to Scenario 5 of Theorem 6.2.
          (Procedure 2)}
18:     end if
19:   end for
20: end for
21: Include the frequent equivalence classes in $\mathcal{F}$ into $\mathcal{F}_{upd+}$;
    return $\mathcal{F}_{upd+}$ and the updated GE-tree;
```

6.1.3 Proposed Algorithm: PSM+

A novel incremental maintenance algorithm, *Pattern Space Maintenance+* (PSM+), is proposed based on the *GE-tree*. The pseudo-code of PSM+ is presented in Algorithm 3, Procedures 2 and 4. In Algorithm 3, Procedures 2 and 4, we use $X.support$ to denote the support of pattern X or equivalence class X; we use $X.close$ to denote the closed pattern of equivalence class X and we use $X.keys$ to denote the set of generators of equivalence class X. We have also proven the correctness of PSM+.

Procedure 4 splitEC

```
Input: $\mathcal{F}$ the set of frequent equivalence classes; $t$ the incremental transaction; and $G$ the
    updating generator.
Output: The updated $\mathcal{F}$.
Method:
 1: Let $EC$ be the equivalence class of $G$ in $\mathcal{F}$;
    {First split out:}
 2: $EC.keys = min\{\{K|K \in EC.keys \wedge K \not\subseteq t\} \cup \{K' \cup \{x_i\}|K' \in EC.keys \wedge K' \subseteq t, x_i \in EC.close \wedge x_i \notin t\}\}$; {$EC.close$ remains the same.}
    {Second split out:}
 3: $C_{new} = EC.close \cap t$;
 4: if $\exists EC'' \in \mathcal{F}$ such that $EC''.close = C_{new}$ then
 5:    $EC''.support = G.support$; {$EC''$ already exists.}
 6:    $G \to EC''.keys$;
 7: else
 8:    Create new equivalence class $EC'$;
 9:    $EC'.close = C_{new}$, $EC'.support = G.support$, $G \to EC'.keys$;
10:    $EC' \to \mathcal{F}$;
11: end if
    return $\mathcal{F}$;
```

Theorem 6.5. *PSM+ presented in Algorithm 3 correctly maintains the frequent pattern space, which is represented using equivalence classes, for incremental updates.*

Proof. According to Theorem 4.16, after the insertion of each new transaction t_+, there are only 5 scenarios for any frequent equivalence class $[P]_{\mathcal{D}_{upd+}}$. We prove the correctness of our algorithm according to these 5 scenarios.

For Scenario 1, suppose (i)$P \in \mathcal{F}(ms_\%, \mathcal{D}_{org})$, (ii) $P \not\subseteq t_+$ and (iii) $Q \not\subseteq t_+$ for all $Q \in [P]_{\mathcal{D}_{org}}$. Point (i) implies that $[P]_{\mathcal{D}_{org}}$ is an existing frequent equivalence class. Then Point (iii) implies that none of the generators of $[P]_{\mathcal{D}_{org}}$ will satisfy the condition in Line 7. As a result, $[P]_{\mathcal{D}_{org}}$ will skip all the maintenance actions and remain unchanged as desired.

For Scenario 2, suppose (i)$P \in \mathcal{F}(ms_\%, \mathcal{D}_{org})$, (ii) $P \subseteq t_+$ and (iii) $Q \subseteq t_+$ for all $Q \in [P]_{\mathcal{D}_{org}}$. Point (iii) implies that the generators of $[P]_{\mathcal{D}_{org}}$ satisfy the condition in Line 7, and the support of the generators will be updated by Line 8. Point (i) implies that $[P]_{\mathcal{D}_{org}}$ is an existing frequent equivalence class. Thus the generators of $[P]_{\mathcal{D}_{org}}$ are existing frequent generators, which satisfy the condition in Line 9. Then Point (iii) also implies that the closed pattern of $[P]_{\mathcal{D}_{org}}$ satisfies the condition in Line 11. Therefore, the support of $[P]_{\mathcal{D}_{org}}$ will be updated in Line 12, but $[P]_{\mathcal{D}_{org}}$ remains unchanged as desired.

For Scenario 3, suppose (i) $P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, (ii) $P \subseteq t_+$ and (iii) $Q \nsubseteq t_+$ for some $Q \in [P]_{\mathcal{D}_{org}}$. Point (ii) implies that some generators of $[P]_{\mathcal{D}_{org}}$ will satisfy the condition in Line 7, and Point (i) implies the condition in Line 9 is also satisfied. Then Point (iii) implies that the condition in Line 11 is not satisfied. Thus the equivalence class will be split into two by Line 14 (Procedure 4) as desired. In particular, $[P]_{\mathcal{D}_{upd+}}$ described in Scenario 3 is updated in Line 3 to 11 of Procedure 4.

For Scenario 4, suppose (i) $P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, (ii) $P \nsubseteq t_+$ and (iii) $Q \subseteq t_+$ for some $Q \in [P]_{\mathcal{D}_{org}}$. Point (iii) implies that some generators of $[P]_{\mathcal{D}_{org}}$ will satisfy the condition in Line 7, and Point (i) implies the condition in Line 9 is also satisfied. Then Point (ii) implies that the condition in Line 11 is not satisfied. Thus the equivalence class will be split into two by Line 14 (Procedure 4) as desired. Being complement to Scenario 3, $[P]_{\mathcal{D}_{upd+}}$ described in Scenario 4 is updated in Line 2 of Procedure 4.

For Scenario 5, suppose (i) $P \notin \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, (ii) $P \subseteq t_+$ and (iii) $Sup(P, \mathcal{D}_{upd+}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd+}|$. For this scenario, we have two cases. In the first case, P is in $\mathcal{D}_{org}$. In this case, the generators of $[P]_{\mathcal{D}_{org}}$ are already included in the *GE-tree*. Therefore, Point (ii) implies that the condition in Line 7 is satisfied. Point (i) then implies that Line 9 is not satisfied. Then we check Line 16. Point (iii) implies that the generators of $[P]_{\mathcal{D}_{upd+}}$ satisfy the condition in Line 16. Therefore, we will go to Line 17 and go into Procedure 2. In Line 3 to 11 of Procedure 2, $[P]_{\mathcal{D}_{upd+}}$ is then constructed and included as a newly emerged frequent equivalence class as desired. In the second case, P is not in $\mathcal{D}_{org}$. In this case, the generators of $[P]_{\mathcal{D}_{org}}$ are not in *GE-tree* yet. Therefore, the new generators will be included by Line 5. Then the generators and the equivalence class are then updated in the same way as in the first case.

Finally, since an incremental update induces the data size and the absolute support threshold to increase, Line 21 is put in to remove equivalence classes that are no longer frequent. With that, the theorem is proven. □

6.1.3.1 A Running Example

We demonstrate how PSM+ (presented in Algorithm 3) updates the frequent pattern space with the example shown in Figure 6.3. In Figure 6.3, the original dataset, $\mathcal{D}_{org}$, consists of 6 transactions; the minimum support threshold $ms_{\%} = 20\%$; and two incremental transactions $\{b, c, d\}$ and $\{a, f\}$ are to be inserted. Therefore, $|\mathcal{D}_{upd+}| = |\mathcal{D}_{org}| + |\mathcal{D}_{inc}| = 8$, and the absolute support threshold is updated by Line 2 of Algorithm 3 as $ms_a = \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil = 2$. For each incremental transaction, PSM+ updates the affected equivalence classes through updating their corresponding generators. In Figure 6.3 (b) and (c), the affected generators and equivalence classes are highlighted in bold.

We further illustrate in details how PSM+ addresses different maintenance scenarios with a few representative examples. First, we investigate the scenario, where the support of the updating equivalence class needs to be increased. Suppose incremental transaction $\{b, c, d\}$ is inserted. Since $\{b, c, d\}$ contains no new items, it does not satisfy the condition of Line 4 in Algorithm 3. Thus, Line 5 is skipped. Then, according to the top-down left-right enumeration and traverse order of *GE-tree*, pattern $\{c\}$ is the first generator that satisfies Line 7 of Algorithm 3 (for $\{c\} \subseteq \{b, c, d\}$). Thus $\{c\}$ is an affected generator. The support of $\{c\}$ is then updated by Line 8. Since $\{c\}$ is an existing frequent generator, Line 9 is satisfied. We then update the corresponding equivalence class of $\{c\}$, $EC2$. As shown in Figure 6.3 (b), the closed pattern of $EC2$ is also $\{c\}$. Thus, we have $EC2.close \subseteq \{b, c, d\}$, and Line 11 is satisfied. The support of $EC2$ is then updated by Line 12, and $EC2$ skips all other update actions as desired.

Second, we investigate the scenario, where the updating equivalence class needs to be split. Still consider the case, where the incremental transaction $\{b, c, d\}$ is inserted. In this case, we use generator $\{d, c\}$ as an example. The support of $\{d, c\}$ is updated in the same way as generator $\{c\}$ in the above example. However, different from generator $\{c\}$, the corresponding equivalence class of $\{d, c\}$ is $EC5$ in Figure 6.3 (a), and, more importantly, $EC5.closed = \{d, c, a\} \not\subseteq \{b, c, d\}$. Therefore, Line 11 of Algorithm 3 is not satisfied. Thus, as desired, $EC5$ will be split into two as described in Procedure 4. As shown in Figure 6.3 (b), $EC5$ splits into $EC5'$ and $EC6'$. In Procedure 4, $EC6'$ is considered as the first split out of $EC5$, and it is updated by Line 2 of Procedure 4. On the other hand, $EC5'$ is considered as the second split out, and it is constructed by Line 3 to 11.

Third, we investigate the scenario, where new frequent generator and equivalence class have emerged. In this case, negative border generator $\{b, c\}$ in Figure 6.3 (a) is used as an example. After the insertion of $\{b, c, d\}$, the support of $\{b, c\}$ is updated in the same manner as the previous two examples. Different from the previous examples, $\{b, c\}$ is not a frequent generator but a negative border generator. As a result, Line 9 in Algorithm 3 is not satisfied. However, as highlighted in Figure 6.3 (b), generator $\{b, c\}$ becomes frequent after the update. Thus, Line 16 is satisfied, and the corresponding equivalence class $EC7$ is then included as frequent equivalence class by Line 1 to 11 of Procedure 2. Furthermore, $\{b, c\}$ also acts as a starting point for further enumeration of new generators as stated in Line 12 to 19 of Procedure 2.

Lastly, we investigate the scenario, where new items are introduced. Incremental transaction $\{a, f\}$ is an good example for this scenario. Different from transaction $\{b, c, d\}$, transaction $\{a, f\}$ consists of new item f. This implies the condition in Line 4 of Algorithm 3 is satisfied. Therefore, as illustrated in Figure 6.3 (c), after the insertion of transaction $\{a, f\}$, generator $\{f\}$ is inserted to the *GE-tree* by Line 5 of Algorithm 3 as desired. Note that the support of $\{f\}$ is first initiated to 0. This is because the support of $\{f\}$ will be then updated by Line 8 as the update goes on.

6.1.3.2 Time Complexity

We have justified the correctness of PSM+ with a theoretical proof and a running example. We now demonstrate that PSM+ is also computationally effective.

The computational complexity of an incremental algorithm is generally measured as a function of the size of areas affected by the data update [62]. In the case of frequent pattern maintenance, the size of affected areas is equivalent to the number of equivalence classes that are affected by the data update. Recall that the major strength of PSM+ is that, with *GE-tree*, PSM+ is able to effectively identify and update only the equivalence classes that are affected by the data update. Equivalence classes that are not affected by the update will be skipped, and thus no unnecessary updates are involved. As a result, the computation involved in PSM+ is proportional to the number of affected equivalence class, $N_{affected}$, and the computation complexity of PSM+ can be expressed as $\mathcal{O}(N_{affected})$.

According to Theorem 6.2, the affected equivalence classes can be categorized into three main types: (I) existing frequent equivalence classes that require only support update; (II) existing frequent equivalence classes that split after the update; and (III) newly frequent equivalence classes that emerge after the update. We observe that the computation required to update these 3 types of affected equivalence classes are very different from one and other. With the help of *GE-tree*, the update of type (I) and type (II) equivalence classes, as demonstrated in Section 6.1.2.2, can be efficiently completed with little computational overhead. On the other hand, the generation of type (III) equivalence classes involves computational expensive processes, which include enumeration and verification of newly frequent generators, generation of corresponding closed patterns and formation of new equivalence classes. Therefore, the generation of type (III) equivalence classes is much more computationally expensive compared to types (I) & (II) equivalence classes, and the computational complexity of PSM+ is mainly contributed by the generation of type (III) equivalence classes. Moreover, in PSM+, the computation required to generate type (III) equivalence classes is proportional to the number of generators enumerated, N_{enum}.

dataset	#PSM+	#FPgrowth*	#GC-growth
BMS-POS ($ms_{\%} = 0.1\%$)	80	110K	110K
BMS-WEBVIEW-1 ($ms_{\%} = 0.1\%$)	250	3K	3K
chess ($ms_{\%} = 40\%$)	350K	6M	1M
connect-4 ($ms_{\%} = 20\%$)	80K	1800M	1M
mushroom ($ms_{\%} = 0.5\%$)	10K	300M	165K
pumsb_star ($ms_{\%} = 30\%$)	2K	400K	27K
retail ($ms_{\%} = 0.1\%$)	270	8K	8K
T10I4D100K ($ms_{\%} = 0.5\%$)	11	1K	1K
T40I10D100K ($ms_{\%} = 10\%$)	7K	70K	55K

Table 6.1: Comparison of the number of patterns enumerated by PSM+, FPgrowth* and GC-growth. Notations: #PSM+, #FPgrowth* and #GC-growth denote the approximated number of patterns enumerated by the respectively algorithms.

Thus, we can approximate the complexity of PSM+ as $O(N_{enum})$.

We have conducted some experiments to compare the number of patterns enumerated by PSM+ with the ones of FPgrowth* and GC-growth. In the experiment, the number of patterns enumerated is recorded for the scenario where the size of new transactions $\mathcal{D}_{inc}$ is 10% of the original data size. The comparison results are summarized in Table 6.1. We observe that the number of patterns enumerated by PSM+ is smaller than the other two by a few orders of magnitude. Therefore, based on computational complexity, PSM+ is much more effective than FPgrowth* and GC-growth.

6.1.3.3 Implementation Details

This section discusses the crucial implementation details of PSM+. Note that: although the following implementation techniques are discussed in the context of PSM+, they are also employed in PSM- and PSM$_{\Delta}$ to facilitate the maintenance process.

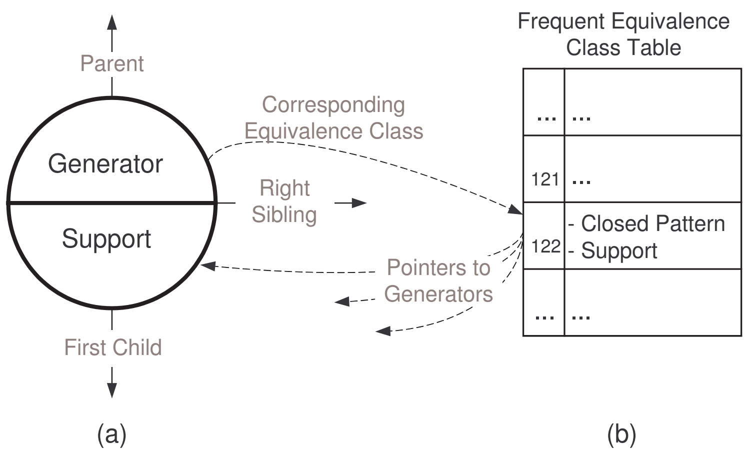


Figure 6.4: (a) Showcase of a *GE-tree* node. (b) The frequent equivalence class table, highlighting the corresponding equivalence class of the *GE-tree* node in (a).

Storage of Frequent Pattern Space

PSM+ takes the original frequent pattern space as input and obtains the updated pattern space by maintaining the original space based on the incremental updates. The frequent pattern space is usually huge. Therefore, effective data structures are needed to compactly store the original and updated pattern spaces. We propose to concisely represent the frequent pattern space with the borders of equivalence classes — closed patterns and generators.

We develop *GE-tree* to compactly store the frequent generators and negative border generators. As shown in Figure 6.3 and 6.4 (a), each node in *GE-tree* stores a generator, and, if the generator is frequent, the node is also linked with its corresponding equivalence class. Frequent equivalence classes, as graphically illustrated in Figure 6.4 (b), are stored in a hash table to achieve fast retrieval. Since each equivalence class is uniquely associated with one closed pattern, frequent equivalence classes are indexed based on their closed patterns. Each bucket in the hash table records the closed pattern and the support value of the associated equivalence class, and it also points to the corresponding generators in the *GE-tree*. Both *GE-tree* and the frequent equivalence class table can be simply constructed with a single scan of the existing frequent equivalence classes, which are represented by its closed patterns and generators.

We employ GC-growth [47] to generate the original frequent pattern space represented in frequent closed patterns and generators. Note that, besides frequent generators, PSM+ also needs the information on negative border generators. As a result, the implementation of GC-growth is modified slightly to generate also the negative border generators. The modification is straightforward: GC-growth just needs to output the points, where the enumeration stops.

Generation of Closed Patterns

GE-tree, with negative border generators, enables effective enumeration of newly emerged frequent generators. To complete the border of equivalence classes, the generation of corresponding closed patterns is required. A prefix tree structure, named *mP-tree*, is developed for this task.

mP-tree is a modification of *P-tree* [34]. *P-tree*, similar to *FP-tree* [33], is a prefix tree that compactly stores transactional datasets. However, unlike *FP-tree*, which contains only frequent items, *P-tree* stores all items in the dataset. *mP-tree* also follows this characteristic. E.g. in Figure 6.5 (b), although item e is not a frequent item, it is still recorded in the *mP-tree*.

The initial *mP-tree* is generated in the same manner as *P-tree*. (Details of the construction of *P-tree* can be referred to [34].) However, *mP-tree* is updated in a

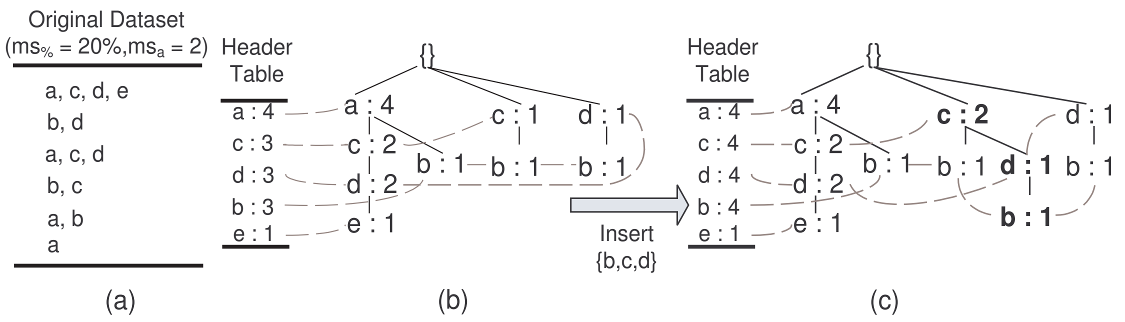


Figure 6.5: (a) A sample data set with $ms_{\%} = 20\%$ and $ms_a = 2$. (b) The *mP-tree* for the dataset in (a). (c) The updated *mP-tree* after the insertion of transaction $\{b, c, d\}$.

different manner. In *P-tree*, new transactions are first inserted into the original tree based on the original item ordering, and, at the same time, the support values of individual items are updated. After that, *P-tree* re-sorts the ordering of items based on their updated support values and restructures the tree based on the updated item ordering. This re-sorting and restructuring process ensures the compactness of *P-tree*, but this process is computational expensive. On the other hand, in *mP-tree*, items are sorted based on their support values in the original dataset. More importantly, the ordering of items remains unchanged for all subsequent updates. This fix ordering of items, as demonstrated in Figure 6.5 (c), allows new transactions to be inserted into *mP-tree* easily without any re-sorting and swapping of tree nodes.

With *mP-tree*, the generation of closed patterns for newly emerged frequent generators becomes straightforward. We use the *mP-tree* in Figure 6.5 (c) as an example. Suppose we want to find out the closed pattern of generator $\{b\}$. We first extract all the branches that consist of generator $\{b\}$ by traversing through the horizontal links (dotted lines in the figure). We then accumulate the counts for all items involved in these branches. In the example, we have item a with count 1, item c with 2, item d

with 2 and item b itself with 4. Since no items have the same count as item b, we can derive that none of them appears in the same transaction as b. Therefore, the closed pattern of generator $\{b\}$ is also $\{b\}$.

In addition, the *mP-tree* is constantly updated as the incremental transactions are inserted. As a result, closed patterns for newly emerged frequent generators can be generated with the *mP-tree* without re-visiting the original dataset.

6.1.4 Experimental Studies

The effectiveness of the incremental maintenance component, PSM+, is experimental tested on the benchmark datasets from the *FIMI* Repository [25]. The statistical information of the benchmark datasets can be referred to Table 5.1. Experiential studies in this chapter are conducted on a PC with 2.4GHz CPU and 3.2G of memory.

In real applications, the size of the incremental dataset $\mathcal{D}_{inc}$ is usually much smaller than the size of the original dataset $\mathcal{D}_{org}$, e.g. a daily sales data vs. an annual sales data, an hourly stock transaction vs. a daily transaction, etc. As a result, the performance of PSM+ is evaluated for $\Delta^{+} \leq 10\%$, where $\Delta^{+} = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

PSM+ is compared with the representative frequent pattern discovery algorithms, FPgrowth* [28] and GC-growth [47]. Recall that, depends on the applications, pattern maintenance can be conducted in two manners: batch maintenance or eager maintenance. As discussed, in eager maintenance, since the pattern space needs to be updated for every update transaction, re-generating the pattern space with discovery algorithms involves heavy redundancy and thus is extremely inefficient. Therefore,

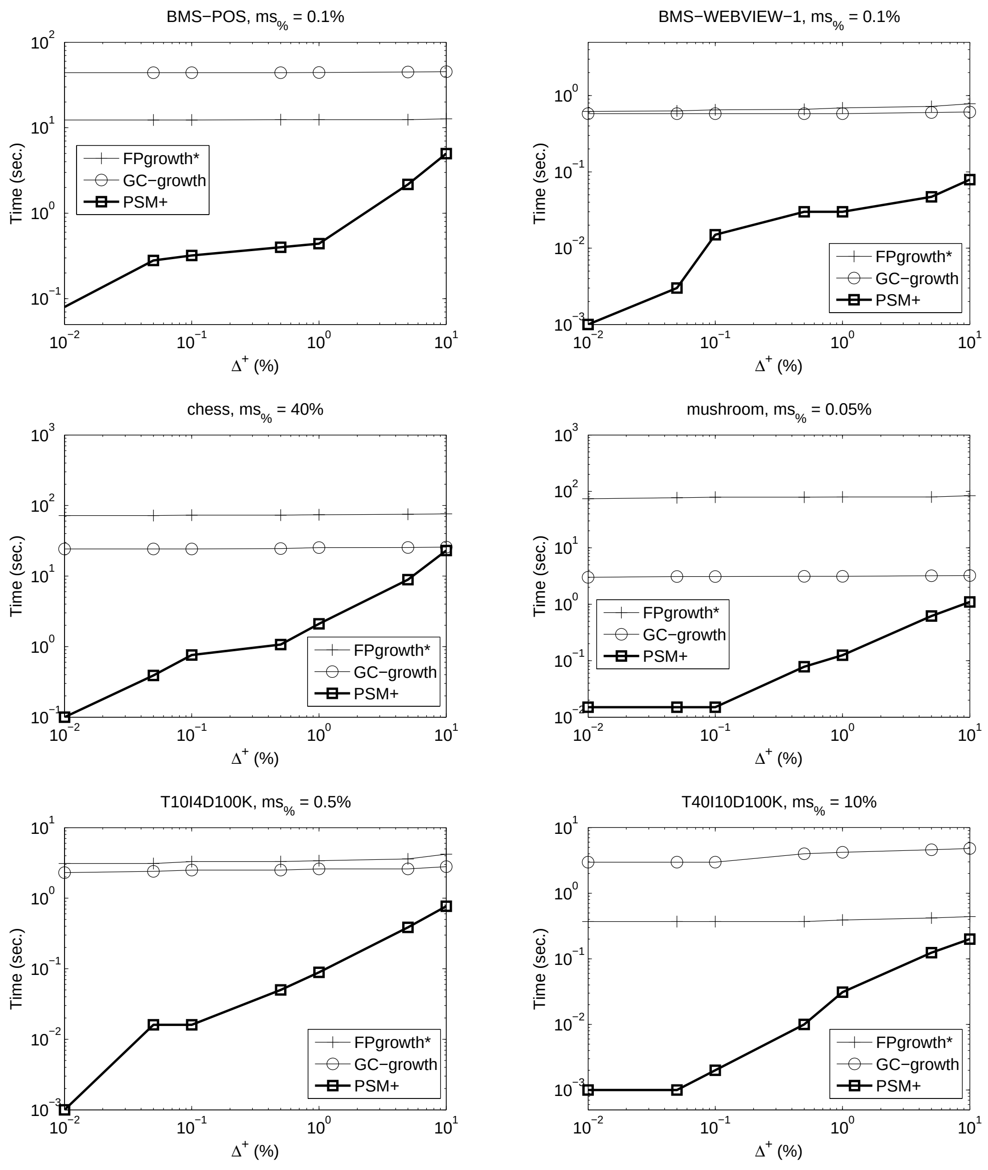


Figure 6.6: Performance comparison of PSM+ and the pattern discovery algorithms: FPgrowth* and GC-growth. Notations: $\Delta^+ = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

updating the pattern space with our proposed algorithm is definitely a better option. Thus, in our experimental studies, we focus only on the batch maintenance scenario.

Figure 6.6 graphically compares the performance of PSM+, FPgrowth* and GC-growth. It can be seen that PSM+ is much faster than both discovery algorithms, especially when the update interval is small. When Δ^+ is below 1%, PSM+ outperforms the discovery algorithms by about 3 orders of magnitude. When Δ^+ is up to 10%, PSM+ is still at least twice faster, and, for the particular dataset *BMS-WEBVIEW-1*, PSM+ is still more than 10 times faster. On the other hand, as can be observed from Figure 6.6, it is inevitable that the advantage of PSM+ shrinks gradually as the update interval increases.

The detailed "speed up" achieved by PSM+ (Equation 5.1) is summarized in Table 6.2. As shown in the table, in the best scenarios, PSM+ is faster than FPgrowth* by more than 3000 times and faster than GC-growth by almost 2000 times; in the worst cases, PSM+ is still about twice faster; and, on average, PSM+ outperforms both discovery algorithms by more than 2 orders of magnitude. In addition, through our experimental studies, we conclude that PSM+ is, in general, more effective than the discovery algorithms as far as the update interval $\Delta^+ \leq 10\%$.

PSM+ is also compared with the state-of-the-art maintenance algorithms, which includes Borders [5], CanTree [44], moment [18] and ZIGZAG [70]. Some representative results are graphically presented in Figure 6.7. According to the empirical results, PSM+ is the most effective algorithm among all. Take dataset *mushroom* as an example. PSM+ is more than 2 order of magnitude faster than CanTree, and, compared with Borders, moment and ZIGZAG, it is faster by 4 orders of magnitude. The average "speed up" of PSM+ against the maintenance algorithms is also summarized in Table 6.2. PSM+, on average, outperforms Borders by more than 4 orders of magnitude, outperforms moment and ZIGZAG by more than 3 orders of magnitude and outperforms CanTree by over 700 times.

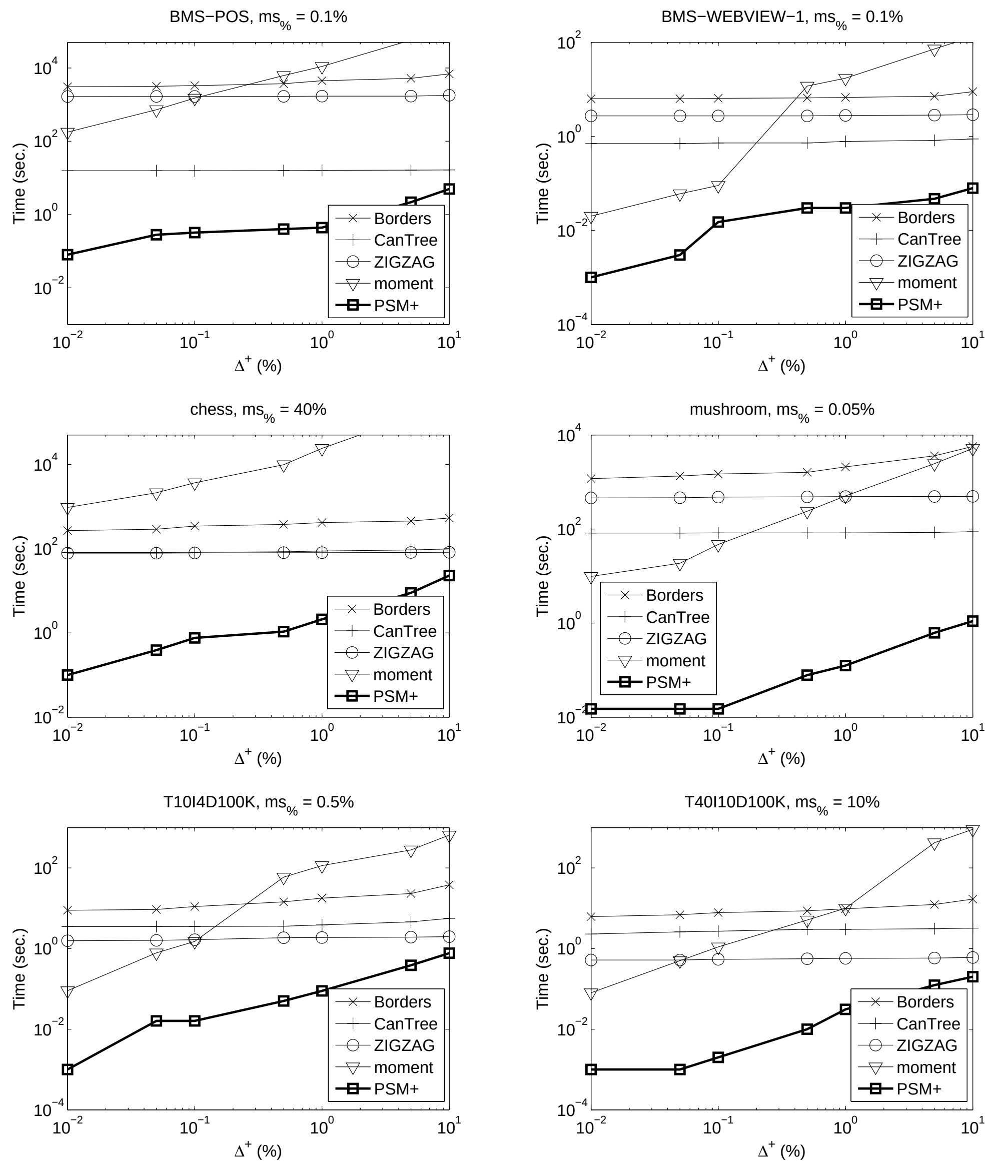


Figure 6.7: Performance comparison of PSM+ and the pattern maintenance algorithms, Borders, CanTree, ZIGZAG and moment. Notations: $\Delta^+ = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

Dataset	Discovery Algorithms		Maintenance Algorithms			
	FPgrowth*	GC-growth	Borders	CanTree	ZIGZAG	moment
accidents ($ms_{\%} = 50\%$)	12.5	76	490	15	270	22
accidents ($ms_{\%} = 40\%$)	1.6	9.5	135	2	56.5	6.2
BMS-POS ($ms_{\%} = 0.1\%$)	43	155	11,900	55	5,880	14,400
BMS-POS ($ms_{\%} = 0.5\%$)	**126**	**390**	**21,000**	**130**	**13,500**	**23,000**
BMS-WEBVIEW-1 ($ms_{\%} = 0.1\%$)	136	125	1,360	152	588	741
BMS-WEBVIEW-1 ($ms_{\%} = 0.05\%$)	963	370	5,200	1,015	672	75
BMS-WEBVIEW-2 ($ms_{\%} = 0.05\%$)	35	96	4,300	40	1,900	715
BMS-WEBVIEW-2 ($ms_{\%} = 0.01\%$)	**1,300**	**316**	**32,200**	**1,420**	**13,000**	**615**
chess ($ms_{\%} = 50\%$)	590	96	3,400	620	1,395	13,000
chess ($ms_{\%} = 40\%$)	169	18	640	180	172	18,100
connect-4 ($ms_{\%} = 50\%$)	2,280	8.2	5,200	2,340	1,400	826
connect-4 ($ms_{\%} = 45\%$)	2,740	5.6	5,300	2,810	1,800	824
mushroom ($ms_{\%} = 0.1\%$)	**3,085**	**380**	**67,000**	**3,121**	**47,800**	**3,216**
mushroom ($ms_{\%} = 0.05\%$)	2,457	81	45,000	2,630	15,000	2,960
pumsb ($ms_{\%} = 70\%$)	1.6	1.5	32	1.8	6.9	1,662
pumsb ($ms_{\%} = 60\%$)	3.5	23.5	56	3.8	16.5	640
pumsb_star ($ms_{\%} = 50\%$)	101	420	273	123	25.7	7,540
pumsb_star ($ms_{\%} = 40\%$)	3.6	20	22	7.2	16	2,970
retail ($ms_{\%} = 0.1\%$)	640	247	36,000	735	27,100	18,210
retail ($ms_{\%} = 0.05\%$)	**985**	**98.5**	**61,000**	**1,050**	**38,500**	**28,340**
T10I4D100K ($ms_{\%} = 0.5\%$)	150	374	1,540	200	261	609
T10I4D100K ($ms_{\%} = 0.05\%$)	41	64	360	45.5	120	81
T40I10D100K ($ms_{\%} = 10\%$)	140	1,145	2,600	955	102	1,415
T40I10D100K ($ms_{\%} = 5\%$)	138	**1,777**	2,160	269	36	1,118
Average	672	262	12,800	746	7,067	5,878

Table 6.2: Average speed up of PSM+ over benchmark datasets for batch maintenance (when $\Delta^{+} \leq 10\%$).

6.2 Decremental Maintenance

This section discusses the decremental maintenance component, PSM-. In the decremental update, some old transactions $\mathcal{D}_{dec}$ are removed from the original dataset $\mathcal{D}_{org}$, and thus the updated dataset $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \mathcal{D}_{dec}$. Given a support threshold, the task of decremental maintenance is to obtain the updated pattern space by maintaining the original pattern space.

We first recap how the space of frequent patterns, which is represented using equivalence classes, evolves under decremental updates. Similar to incremental maintenance, we simplify the update scenarios by considering one decremental transaction at a time. Based on the space evolution study, we then summarize the major computational tasks in decremental maintenance. We further demonstrate how these computational tasks can also be completed efficiently using *GE-tree*. PSM- is then studied both algorithmically and experimentally.

6.2.1 Evolution of Pattern Space

There is an obvious duality between incremental and decremental updates. They are basically reverse operations of each other. Therefore, similar to incremental updates, existing equivalence classes may also evolves in three different ways under decremental updates: existing classes may remain unchanged without any change in support; or they may remain unchanged but with an decreased support; or they may expand by merging with other classes. Since the support of equivalence classes may decrease after an decremental update, some existing frequent equivalence classes may become infrequent.

Under incremental updates, one existing equivalence class may split into multiple classes, and, reversely, multiple classes may merge into one under decremental updates. We have successfully simplified the class splitting scenarios under incremental updates by considering one incremental transaction at a time. Similarly, the class merging scenarios under decremental updates can also be simplified using the same strategy.

Proposition 6.6. *Let $[P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{org}}$ be two existing equivalence classes in $\mathcal{D}_{org}$, where $[P]_{\mathcal{D}_{org}} \cap [Q]_{\mathcal{D}_{org}} = \emptyset$. Also let $t-$ be a decremental transaction and $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \{t-\}$. In $\mathcal{D}_{upd-}$, $[P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{org}}$ merge into $[P]_{\mathcal{D}_{upd-}}$, only if (i) $P \nsubseteq t- \ \wedge \ Q \subseteq t-$ or (ii) $P \subseteq t- \ \wedge \ Q \nsubseteq t-$.*

Proof. By normal definition, $[P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{org}}$ merge into $[P]_{\mathcal{D}_{upd-}}$ means $[P]_{\mathcal{D}_{org}} \subset [P]_{\mathcal{D}_{upd-}}$ and $[Q]_{\mathcal{D}_{org}} \subset [P]_{\mathcal{D}_{upd-}}$. This further means $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$. Since $[P]_{\mathcal{D}_{org}} \cap [Q]_{\mathcal{D}_{org}} = \emptyset$, we have $f(P, \mathcal{D}_{org}) \neq f(Q, \mathcal{D}_{org})$. For any pattern P and Q, there are 4 possibilities: (1) $P \nsubseteq t-$, $Q \nsubseteq t-$, (2) $P \nsubseteq t-$, $Q \subseteq t-$, (3) $P \subseteq t-$, $Q \nsubseteq t-$, and (4) $P \subseteq t-$, $Q \subseteq t-$. Case (2) and (3) correspond to conditions (i) and (ii) in the proposition. Moreover, Case (2) and (3) are equivalent, because any assignment for P and Q in Case (2) can be oppositely assigned in Case (3). For both Case (1) and (4), we have $f(P, \mathcal{D}_{dec}) = f(Q, \mathcal{D}_{dec})$. Since $f(P, \mathcal{D}_{org}) \neq f(Q, \mathcal{D}_{org})$, this implies that $f(P, \mathcal{D}_{org}) - f(P, \mathcal{D}_{dec}) \neq f(Q, \mathcal{D}_{org}) - (Q, \mathcal{D}_{dec})$. Therefore, $f(P, \mathcal{D}_{upd-}) \neq f(Q, \mathcal{D}_{upd-})$ for Case (1) and (4). This proves that, for $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$ and $[P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{org}}$ merge into $[P]_{\mathcal{D}_{upd-}}$, one of Case (2) and (3) must be true. $\square$

Since conditions (i) and (ii) in Proposition 6.6 are equivalent, without loss of generality, we assume condition (i) is true for the following discussion. Following Proposition 6.6, we have Proposition 6.7, as the duality of Proposition 6.1, for decremental updates. We use $Close(X)$ to denote the closed pattern of equivalence class X and $Keys(X)$ to denote the generators of the class.

Proposition 6.7. *Let $[P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{org}}$ be two existing equivalence classes in $\mathcal{D}_{org}$, where $[P]_{\mathcal{D}_{org}} \cap [Q]_{\mathcal{D}_{org}} = \emptyset \ \wedge \ P \not\subseteq t- \ \wedge \ Q \subseteq t-$. Also let $t-$ be a decremental transaction and $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \{t-\}$. In $\mathcal{D}_{upd-}$, $[P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{org}}$ merge into $[P]_{\mathcal{D}_{upd-}}$ iff $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$.*

Moreover, $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$ iff $f(Q, \mathcal{D}_{org}) - f(P, \mathcal{D}_{org}) = \{t-\}$.

Then, $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup [Q]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$, $Close([P]_{\mathcal{D}_{upd-}}) = Close([P]_{\mathcal{D}_{org}})$ and $Keys([P]_{\mathcal{D}_{upd-}}) = \min\{K | K \in Keys([P]_{\mathcal{D}_{org}}) \vee K \in Keys([Q]_{\mathcal{D}_{org}})\}$

Proof. The first part of the proposition is obvious given the definition of equivalence class (Definition 4.3).

Next, we prove the second part. For the left-to-right direction, suppose $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$. Since $P \not\subseteq t-$, $f(P, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{org})$. Also since $Q \subseteq t-$, $f(Q, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{org}) - \{t-\}$. This implies that $f(P, \mathcal{D}_{org}) = f(Q, \mathcal{D}_{org}) - \{t-\}$. Thus $f(Q, \mathcal{D}_{org}) - f(P, \mathcal{D}_{org}) = \{t-\}$. The left-to-right direction is proven. For the right-to-left direction, suppose (i) $P \not\subseteq t-$ and $Q \subseteq t-$ and (ii) $f(Q, \mathcal{D}_{org}) - f(P, \mathcal{D}_{org}) = \{t-\}$. Point (i) and (ii) directly implies that

$f(Q, \mathcal{D}_{org}) - \{t-\} = f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{org}) = f(P, \mathcal{D}_{upd-})$. The right-to-left direction is proven.

For the last part of the proposition, we first prove $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup [Q]_{\mathcal{D}_{org}}$. Suppose $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$. Then, according to the definition of equivalence class, $[P]_{\mathcal{D}_{upd-}} = [Q]_{\mathcal{D}_{upd-}}$. Also according to Corollary 4.17, we have $[P]_{\mathcal{D}_{org}} \subseteq [P]_{\mathcal{D}_{upd-}}$ and $[Q]_{\mathcal{D}_{org}} \subseteq [Q]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{upd-}}$. Therefore, we have $[P]_{\mathcal{D}_{upd-}} \supseteq [P]_{\mathcal{D}_{org}} \cup [Q]_{\mathcal{D}_{org}}$. Let $\mathcal{O} = [P]_{\mathcal{D}_{org}} \cup [Q]_{\mathcal{D}_{org}}$ and assume $\exists X \notin \mathcal{O}$ but $X \in [P]_{\mathcal{D}_{upd-}}$. Since $X \in [P]_{\mathcal{D}_{upd-}}$, $f(P, \mathcal{D}_{upd-}) = f(X, \mathcal{D}_{upd-})$. According to the second part of the proposition, we have $f(X, \mathcal{D}_{org}) - f(P, \mathcal{D}_{org}) = \{t-\}$. This further implies that $f(X, \mathcal{D}_{org}) = f(Q, \mathcal{D}_{org})$ and $X \in [Q]_{\mathcal{D}_{org}}$. This contradicts with our initial assumption. Thus $\nexists X \notin \mathcal{O}$ but $X \in [P]_{\mathcal{D}_{upd-}}$. Therefore $[P]_{\mathcal{D}_{upd-}} = \mathcal{O} = [P]_{\mathcal{D}_{org}} \cup [Q]_{\mathcal{D}_{org}}$.

Next we prove $Close([P]_{\mathcal{D}_{upd-}}) = Close([P]_{\mathcal{D}_{org}})$. Let $C = Close([P]_{\mathcal{D}_{org}})$ and assume that there exists pattern $C' \supset C$ that $C' \in [P]_{\mathcal{D}_{upd-}}$. Since C is the closed pattern of $[P]_{\mathcal{D}_{org}}$ and $C' \supset C$, according to Definition 2.3, we know $C' \notin [P]_{\mathcal{D}_{org}}$ and $f(C', \mathcal{D}_{org}) \neq f(P, \mathcal{D}_{org})$. Also since $P \nsubseteq t-$, $C \nsubseteq t-$ ($C \in [P]_{\mathcal{D}_{org}}$) and $C' \nsubseteq t-$ ($C' \supset C$). Therefore, $f(C', \mathcal{D}_{upd-}) = f(C', \mathcal{D}_{org})$. Combining the facts that $f(C', \mathcal{D}_{org}) \neq f(P, \mathcal{D}_{org})$, $f(P, \mathcal{D}_{org}) = f(P, \mathcal{D}_{upd-})$ and $f(C', \mathcal{D}_{upd-}) = f(C', \mathcal{D}_{org})$, we have $f(C', \mathcal{D}_{upd-}) \neq f(P, \mathcal{D}_{upd-})$ and $C' \notin [P]_{\mathcal{D}_{upd-}}$. This contradicts with the initial assumption. Thus we can conclude that $C' \notin [P]_{\mathcal{D}_{upd-}}$ for all $C' \supset C$. According to Fact 4.6, C is the closed pattern of $[P]_{\mathcal{D}_{upd-}}$.

For $Keys([P]_{\mathcal{D}_{upd-}})$, the formula basically follows the definition of generator in Fact 4.6. Also since $P \nsubseteq t-$, we have $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$. □

We can also simplify the evolution scenarios in Theorem 4.20 as follows.

Theorem 6.8. *Let $\mathcal{D}_{org}$ be the original dataset, $t-$ be the decremental transaction, $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \{t-\}$ be the updated dataset and $ms_{\%}$ be the support threshold. For every frequent equivalence class $[P]_{\mathcal{D}_{org}}$ in $\mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, exactly one of the 5 scenarios below holds:*

1. *$P \not\subseteq t-$ and there does not exists Q such that $Q \notin [P]_{\mathcal{D}_{org}}$ but $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$, corresponding to the scenario where the equivalence class remains totally unchanged. In this case, $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$ and $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$.*

2. *$P \not\subseteq t-$ and $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$ for some $Q \notin [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where the equivalence class of Q has to merge into the equivalence class of P. Moreover, $[P]_{\mathcal{D}_{upd-}} = [Q]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup [Q]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$, $Close([P]_{\mathcal{D}_{upd-}}) = Close([P]_{\mathcal{D}_{org}})$ and $Keys([P]_{\mathcal{D}_{upd-}}) = min\{K | K \in Keys([P]_{\mathcal{D}_{org}}) \vee K \in Keys([Q]_{\mathcal{D}_{org}})\}$. Furthermore, $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$.*

3. *$P \subseteq t-$ and $sup(P, \mathcal{D}_{upd-}) < \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$, corresponding to the scenario where an existing frequent equivalence class becomes infrequent. In this case, $[P]_{\mathcal{D}_{org}} \notin \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$.*

4. *$P \subseteq t-$, $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ and there does not exists Q such that $Q \notin [P]_{\mathcal{D}_{org}}$ but $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$, corresponding to the scenario where the equivalence class remains the same but with decreased support. In*

this case, $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org}) - 1$ and $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$.

5. *$P \subseteq t-$, $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ and $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$ for some $Q \notin [P]_{\mathcal{D}_{org}}$, corresponding to the scenario where the equivalence class of P has to merge into the equivalence class of Q. This scenario is complement to Scenario 2. In this case, the equivalence class, support, generators, and closed pattern of $[P]_{\mathcal{D}_{upd-}}$ is same as that of $[Q]_{\mathcal{D}_{upd-}}$, as computed in Scenario 2.*

Proof. Scenarios 1 and 3 are obvious. Scenario 2 follows Proposition 6.7, and $[P]_{\mathcal{D}_{org}}$ in Scenario 2 is equivalent to $[P]_{\mathcal{D}_{org}}$ in Proposition 6.7. Scenario 5 is the complementary of Scenario 2. Scenario 5 is also extended from Proposition 6.7, and $[P]_{\mathcal{D}_{org}}$ in Scenario 5 is equivalent to $[Q]_{\mathcal{D}_{org}}$ in Proposition 6.7.

We then prove Scenario 4. Suppose (i) $P \subseteq t-$, (ii) $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ and (iii) there does not exists Q such that $Q \notin [P]_{\mathcal{D}_{org}}$ but $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$. Point (ii) implies that $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$. According to Corollary 4.17, every member of $[P]_{\mathcal{D}_{org}}$ remains to be in $[P]_{\mathcal{D}_{upd-}}$ after the update. Moreover, point (iii) implies that $f(Q, \mathcal{D}_{upd-}) \neq f(P, \mathcal{D}_{upd-})$ for every pattern $Q \notin [P]_{\mathcal{D}_{org}}$. This means no new members will be included into $[P]_{\mathcal{D}_{upd-}}$. Therefore, $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$ and $sup(P, \mathcal{D}_{upd-}) = |f(P, \mathcal{D}_{upd-})| = |f(P, \mathcal{D}_{org}) - \{t-\}| = sup(P, \mathcal{D}_{org}) - 1$.

Last we prove that the theorem is complete. For patterns $P \not\subseteq t-$, it is obvious that Scenario 1 and 2 enumerated all possible cases. For patterns $P \subseteq t-$, it is also obvious that Scenarios 3 to 5 enumerated all possible cases. Therefore, the theorem is complete and correct. □

Theorem 6.8 summarizes how the frequent pattern space evolves after one existing transaction is removed from the original dataset. The theorem also describes how the updated frequent equivalence classes can be derived from the existing frequent equivalence classes.

In addition, opposite to the incremental update, the decremental update decreases the absolute support threshold if the support threshold is initially defined in terms of percentage. This decrease in the absolute support threshold induces new frequent equivalence classes to emerge.

Combining all the above observations, we summarize that the decremental maintenance of the frequent pattern space involves four **computational tasks**: (1) update the support of existing frequent equivalence classes; (2) merge equivalence classes that satisfy Scenarios 2 and 5 of Theorem 6.8; (3) discover newly emerged frequent equivalence classes; and (4) remove existing frequent equivalence classes that are no longer frequent. Task (3) is not needed if the minimum support threshold is defined in terms of absolute count. Task (4) is excluded from our discussion, for its solution is straightforward. Assume that the support threshold is defined in terms of percentage, we here focus on the first three tasks, and we name them respectively as the **Support Update** task, **Class Merging** task and **New Class Discovery** task.

6.2.2 Maintenance of Pattern Space

Decremental updates are the reverse operation of incremental updates, and, analogously, decremental maintenance is the reverse process of incremental maintenance.

As a result, the maintenance data structure *GE-tree* can also be employed to address the computational tasks in decremental maintenance.

For the **Support Update** task in decremental maintenance, it is basically the reverse operation of the one in incremental maintenance. Therefore, the support of existing frequent equivalence classes can be updated using *GE-tree* in the same manner described in Section 6.1.2.2. Except that, as shown in Figure 6.8, in decremental maintenance, the support is decremented instead.

For the **New Class Discovery** task, newly emerged frequent equivalence classes and generators can also be effectively enumerated based on the concept of negative generator border. Details of the enumeration method is presented in Procedure 2 in Section 6.1.2.3. Same as in incremental maintenance, the negative generator border is updated after the removal of each old transactions. However, as illustrated in Figure 6.8 (c), different from incremental updates, when old transactions are removed, the negative generator border shrinks and move towards the root of *GE-tree*.

For the **Class Merging** task, unfortunately, it can not be handled in the same way as the *Class Splitting* task in incremental maintenance. However, extended from Proposition 6.7, we have the following corollary.

Corollary 6.9. *Let $[P]_{\mathcal{D}_{org}}$ and $[Q]_{\mathcal{D}_{org}}$ be two equivalence classes in $\mathcal{D}_{org}$ such that $[P]_{\mathcal{D}_{org}} \cap [Q]_{\mathcal{D}_{org}} = \emptyset$, $P \notin \mathcal{D}_{dec}$ but $Q \in \mathcal{D}_{dec}$. Then $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$, meaning $[P]_{\mathcal{D}_{org}}$ merges with $[Q]_{\mathcal{D}_{org}}$ in $\mathcal{D}_{upd-}$, iff (1) $sup(P, \mathcal{D}_{upd-}) = sup(Q, \mathcal{D}_{upd-})$ and (2) $Close([P]_{\mathcal{D}_{org}}) \supset Close([Q]_{\mathcal{D}_{org}})$.*

Proof. We first prove the left-to-right direction. Suppose (i) $P \notin \mathcal{D}_{dec}$, (ii) $Q \in$

$\mathcal{D}_{dec}$ and (iii) $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$. Point (iii) implies that $sup(P, \mathcal{D}_{upd-}) = sup(Q, \mathcal{D}_{upd-})$. Combining Point (i),(ii) and (iii), we have $f(P, \mathcal{D}_{org}) = f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{org}) - f(Q, \mathcal{D}_{dec})$. This implies that $f(P, \mathcal{D}_{org}) \subset f(Q, \mathcal{D}_{org})$. Therefore, $Close([P]_{\mathcal{D}_{org}}) \supset Close([Q]_{\mathcal{D}_{org}})$ (Fact 4.8).

We then prove the right-to-left direction. Suppose (i) $sup(P, \mathcal{D}_{upd-}) = sup(Q, \mathcal{D}_{upd-})$ and (ii) $Close([P]_{\mathcal{D}_{org}}) \supset Close([Q]_{\mathcal{D}_{org}})$. Point (ii) implies that $f(P, \mathcal{D}_{upd-}) \subseteq f(Q, \mathcal{D}_{upd-})$ (Fact 4.8). Combining this with Point (i), we have $f(P, \mathcal{D}_{upd-}) = f(Q, \mathcal{D}_{upd-})$ as desired. The corollary is proven. □

Corollary 6.9 provides us a means to determine which two equivalence classes need to be merged after a decremental update. Based on Corollary 6.9, one way to handle the *Class Merging* task effectively is to first group the equivalence classes based on their support. This can be done efficiently using a hash table with support values as hash keys. Then, within the group of equivalence classes that shared the same support, we further compare their closed patterns. two equivalence classes are to be merged together, if their closed patterns are superset and subset to each other.

6.2.3 Proposed Algorithm: PSM-

The pseudo-code of PSM- is presented in Algorithm 5 and Procedure 2. In Algorithm 5 and Procedure 2, we use $X.support$ to denote the support of pattern X or equivalence class X; we use $X.close$ to denote the closed pattern of equivalence class X and we use $X.keys$ to denote the set of generators of equivalence class X. We have also proven the correctness of PSM-.

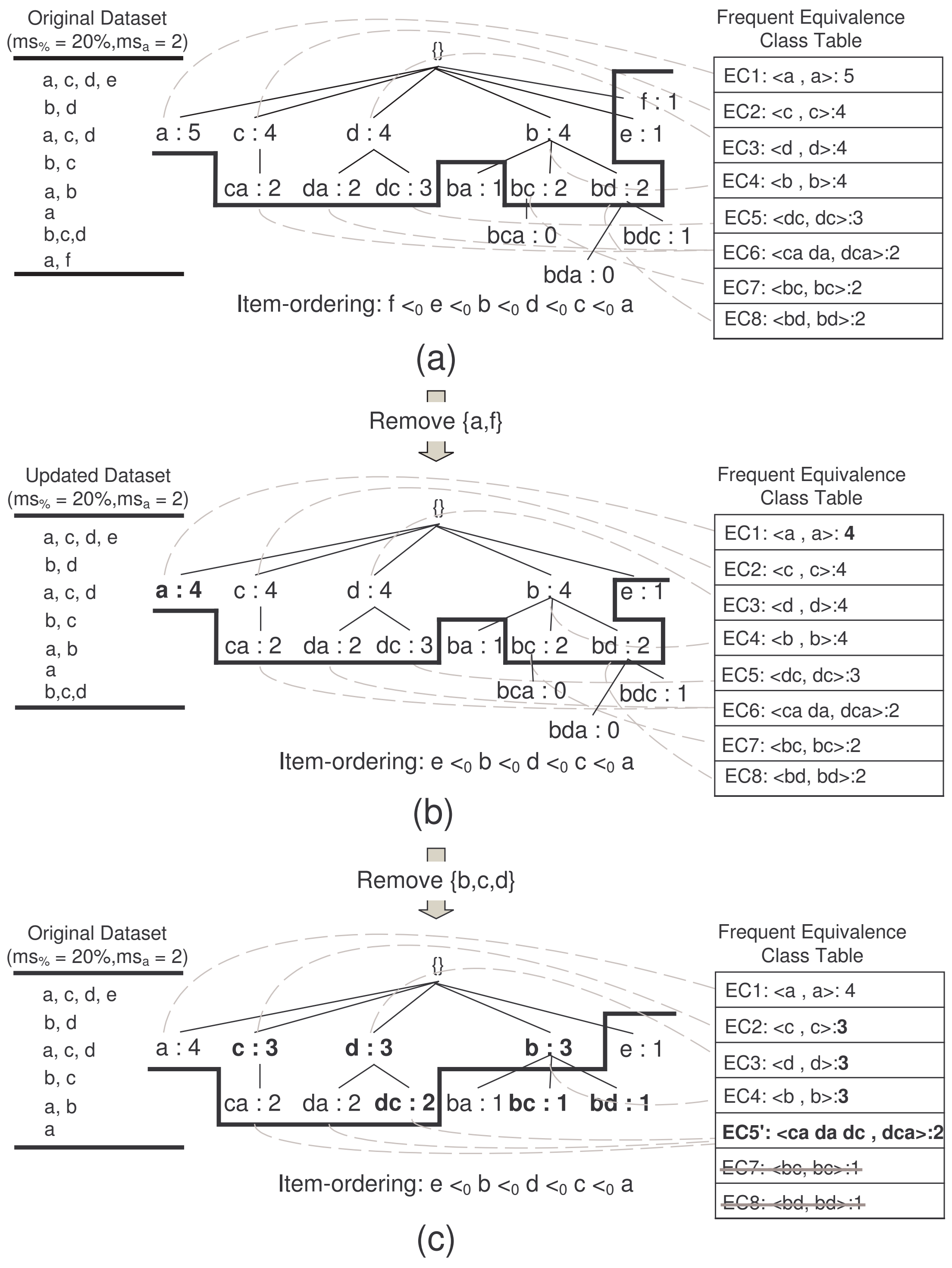


Figure 6.8: (a) The *GE-tree* for the original dataset. (b) The updated *GE-tree* after the existing transaction $\{a, f\}$ is removed. (c) The updated *GE-tree* after the existing transaction $\{b, c, d\}$ is also removed.

Algorithm 5 PSM-

Input: $\mathcal{D}_{dec}$ the decremental dataset; $|\mathcal{D}_{upd-}|$ the size of the updated dataset; $\mathcal{F}_{org}$ the original frequent pattern space represented using equivalence classes ; *GE-tree* and $ms_{\%}$ the support threshold.

Output: $\mathcal{F}_{upd-}$ the updated frequent pattern space represented using equivalence classes and the updated *GE-tree*.

Method:

```
1: $\mathcal{F} := \mathcal{F}_{org}$; {Initialization.}
2: $ms_a = \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$;
3: for all transaction $t$ in $\mathcal{D}_{dec}$ do
4:    for all generator $G$ in GE-tree that $G \subseteq t$ do
5:       $G.support := G.support - 1$;
6:       if $G$ is an existing frequent generator then
7:          Let $EC$ be the equivalence class of $G$ in $\mathcal{F}$;
            {Update the support of existing frequent equivalence classes.}
8:          $EC.support := G.support$;
9:       end if
10:      if $G.support < ms_a$ then
11:         $G \rightarrow$ GE-tree.ngb; {Update the negative generator border.}
12:         Remove all children of $G$ from GE-tree.ngb;
13:      end if
14:   end for
15: end for
16: for all $NG \in$ GE-tree.ngb that $NG.support \geq ms_a$ do
17:    enumNewEC($NG$, $\mathcal{F}$, $ms_a$, GE-tree); {Enumerate new frequent equivalence classes.}
18: end for
19: for all equivalence class $EC \in \mathcal{F}$ do
20:    if $EC.support \geq ms_a$ then
21:       if $\exists EC'$ such that $EC'.support = EC.support$ and $EC.close \subset EC'.close$ then
22:          $EC'.keys = min\{K | K \in EC.keys \wedge K \in EC'.keys\}$;
            {Merging of equivalence classes.}
23:          Remove $EC$ from $\mathcal{F}$;
24:       end if
25:    else
26:       Remove $EC$ from $\mathcal{F}$;
27:    end if
28: end for
29: $\mathcal{F}_{upd-} := \mathcal{F}$
    return $\mathcal{F}_{upd-}$ and the updated GE-tree;
```

Theorem 6.10. *PSM- presented in Algorithm 5 correctly maintains the frequent pattern space, which is represented using equivalence classes, for decremental updates.*

Proof. According to Theorem 6.8, after removing an existing transaction $t-$, an existing frequent equivalence class $[P]_{\mathcal{D}_{org}}$ may evolve in only 5 scenarios. We prove the correctness of our algorithm according to these 5 scenarios.

For Scenario 1, suppose (i) $P \not\subseteq t-$ and (ii) there does not exists Q such that $Q \notin [P]_{\mathcal{D}_{org}}$ but $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$. In Line 1, $[P]_{\mathcal{D}_{org}}$ is included into $\mathcal{F}$ as initialization. Then Point (i) implies that the condition in Line 4 will not be satisfied. Thus, Lines 5 to 15 will be skipped, and the support of $[P]_{\mathcal{D}_{org}}$ remains unchanged as desired. Also since $[P]_{\mathcal{D}_{org}} \in \mathcal{F}(\mathcal{D}_{org}, ms_{\%})$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org}) \geq \lceil ms_{\%} \times |\mathcal{D}_{org}| \rceil \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$. Therefore, the condition in Line 20 is satisfied. Point (ii) implies that Line 21 can not be true (Corollary 6.9). As a result, $[P]_{\mathcal{D}_{org}}$ is included in $\mathcal{F}_{upd-}$ unchanged in Line 29 as desired.

For Scenario 2, suppose (i) $P \not\subseteq t-$ and (ii) $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$ for some $Q \notin [P]_{\mathcal{D}_{org}}$. In Line 1, $[P]_{\mathcal{D}_{org}}$ is included into $\mathcal{F}$ as initialization. Same as in Scenario 1, because of Point (i), the condition in Line 4 is not satisfied, and thus Lines 5 to 15 are skipped. The support of $[P]_{\mathcal{D}_{org}}$ remains unchanged as desired. With the same reasoning in Scenario 1, Line 20 will be true. Point (i) also implies that Line 21 cannot be true. However, Point (ii) implies that there exists equivalence class EC' that satisfies Line 21 and will merge with $[P]_{\mathcal{D}_{org}}$. For the case when more than one transaction is removed, the for-loop between Lines 19 to 28 iteratively merge all such EC's with $[P]_{\mathcal{D}_{org}}$ to form $[P]_{\mathcal{D}_{upd-}}$ as desired. Finally, $[P]_{\mathcal{D}_{upd-}}$ is included in $\mathcal{F}_{upd-}$ in Line 29 as desired.

For Scenario 3, suppose (i) $P \not\subseteq t-$ and (ii) $sup(P, \mathcal{D}_{upd-}) < \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$. As usual, $[P]_{\mathcal{D}_{org}}$ is included into $\mathcal{F}$ as initialization. Point (ii) implies that Line 20 will not be true. Therefore, $[P]_{\mathcal{D}_{org}}$ will be removed from $\mathcal{F}$ in Line 26, and it will not be included in $\mathcal{F}_{upd-}$ as desired.

For Scenario 4, suppose (i) $P \subseteq t-$, (ii) $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ and (iii) there does not exists Q such that $Q \notin [P]_{\mathcal{D}_{org}}$ but $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$. As usual, $[P]_{\mathcal{D}_{org}}$ is included into $\mathcal{F}$ as initialization. Point (i) implies that the condition in Line 4 will be satisfied. Thus the support of $[P]_{\mathcal{D}_{org}}$ will be updated as desired by Line 8. Point (ii) then implies that Line 10 is not true, and thus Lines 11 to 12 are skipped. Point (ii) and (iii) also implies that Line 20 will be true but Line 21 will not be true (Corollary 6.9). As a result, $[P]_{\mathcal{D}_{org}}$ will be include in $\mathcal{F}_{upd-}$ with a updated support as desired.

For Scenario 5, since it is complement to Scenario 2, patterns of Scenario 5 will also be correctly updated as explained in Scenario 2.

Finally, since a decremental update causes the data size and the absolute support threshold to drop, new frequent equivalence classes may emerge. In PSM-, all the newly emerged frequent equivalence classes will be enumerated from the negative generator border by Line 17. With that, the theorem is proven. □

Similar to PSM+, the major contribution to the time complexity of PSM- comes from the *New Class Discovery* task. For the *New Class Discovery* task, the computational complexity is proportional to the number of patterns enumerated. As a result, the time complexity of PSM- can also be approximated as $O(N_{enum})$, where N_{enum} is the number of patterns enumerated. Moreover, the number of patterns need to be enumerated is proportional to the number of newly emerged frequent equivalence classes. In general, under decremental updates, the number of newly emerged frequent equivalence classes is much smaller than the total number of frequent equivalence classes. This theoretically demonstrates that maintaining the frequent pattern space with PSM- is definitely much more effective than re-discovering the pattern space.

6.2.4 Experimental Studies

The effectiveness of PSM- is evaluated through experimental studies. Similar to incremental updates, in real applications, the size of decremental updates is usually much smaller compared to the size of the original dataset. Therefore, in our experimental studies, the performance of PSM- is evaluated for $\Delta^- \leq 10\%$, where $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$. PSM- is compared with the representative pattern discovery algorithms, FPgrowth* [28] and GC-growth [47], for batch maintenance. PSM- is also compared with the state-of-the-art maintenance algorithms that address decremental updates. These algorithms include moment [18] and ZIGZAG [70]. In addition, the performance of PSM- is also compared with our proposed algorithm, TRUM. The representative results are summarized in Figure 6.9.

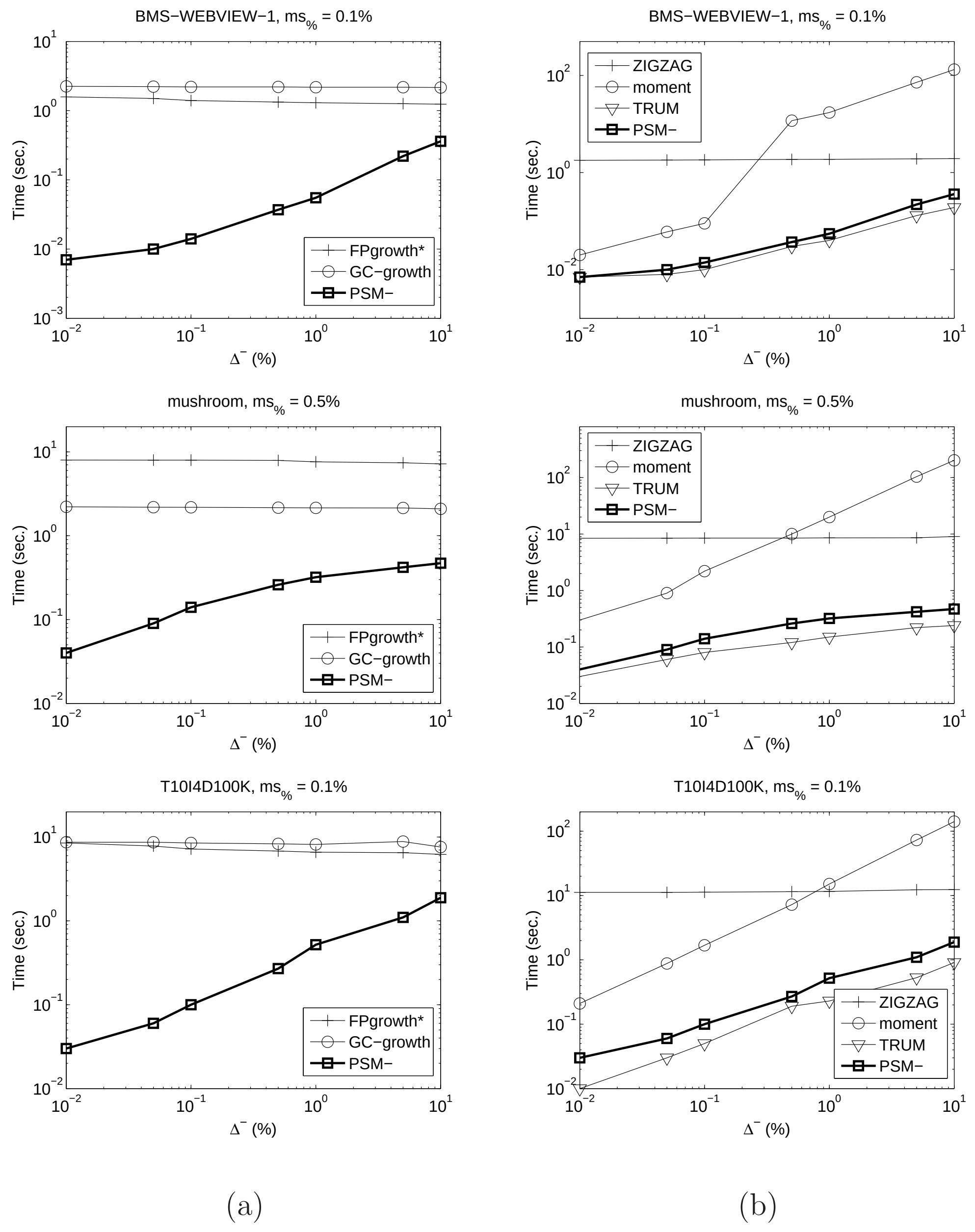


Figure 6.9: (a) Performance comparison of PSM- and the pattern discovery algorithms: FPgrowth* and GC-growth. (b) Performance comparison of PSM- and the pattern maintenance algorithms: ZIGZAG, moment and TRUM. Notations: $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$.

As illustrated in Figure 6.9 (a), PSM- is much more efficient than the discovery algorithms. When the update interval, Δ^-, is below 1%, PSM- outperforms the discovery algorithms by around 2 orders of magnitude; and, when Δ^- is up to 10%, PSM- is still 5 times more efficient. Table 6.3 summarizes the average "speed up" achieved by PSM- (Equation 5.1). Compared with FPgrowth*, PSM- achieves the highest speed up over dataset *mushroom*, where PSM- runs almost 2000 times faster. Compared with GC-growth, PSM- tops on datasets *BMS-POS* and $T10I4D100K$, where PSM- runs over 300 times faster. On average, PSM- outperforms both discovery algorithms by around 2 orders of magnitude. Moreover, similar to other maintenance case, we also observe that the advantage of PSM- diminishes slowly as more transactions are removed from the original dataset. However, based on our experimental results, we are confident that PSM- is more effective compared to the discovery algorithms as far as $\Delta^- \leq 10\%$.

Figure 6.9 (b) graphically compares PSM- with other maintenance algorithms. Compared with moment and ZIGZAG, PSM-, in most cases, is at least 10 times faster. According to Table 6.3, PSM- outperforms ZIGZAG by almost 800 times and outperforms moment by almost 4 orders of magnitude. However, it is also observed that PSM- is not as fast as our proposed decremental maintainer, TRUM. This shows that the *TID-tree* in TRUM is a more effective data structure for decremental maintenance than the *GE-tree* in PSM-. However, *GE-tree* is a more versatile structure that can be applied to various types of updates.

Dataset	Discovery Algorithms		Discovery Algorithms	
	FPgrowth*	GC-growth	ZIGZAG	moment
accidents ($ms_{\%} = 50\%$)	8.5	65	180	15
accidents ($ms_{\%} = 40\%$)	1.5	9	44	4.7
BMS-POS ($ms_{\%} = 0.1\%$)	40	98	5,100	12,000
BMS-POS ($ms_{\%} = 0.01\%$)	**105**	**326**	**10,500**	**21,000**
BMS-WEBVIEW-1 ($ms_{\%} = 0.1\%$)	4.6	40	3.8	150
BMS-WEBVIEW-1 ($ms_{\%} = 0.05\%$)	5.3	45	14.6	187
BMS-WEBVIEW-2 ($ms_{\%} = 0.05\%$)	11.8	24	53	210
BMS-WEBVIEW-2 ($ms_{\%} = 0.01\%$)	9.5	22	48	198
chess ($ms_{\%} = 50\%$)	37	7.6	50	2,800
chess ($ms_{\%} = 40\%$)	102	10	22	**88,000**
connect-4 ($ms_{\%} = 50\%$)	110	2.1	116	1,080
connect-4 ($ms_{\%} = 45\%$)	18	1.3	7.5	170
mushroom ($ms_{\%} = 0.5\%$)	135	44	140	7,200
mushroom ($ms_{\%} = 0.1\%$)	**1,850**	**69**	**432**	**23,400**
pumsb ($ms_{\%} = 70\%$)	4.5	6.6	1.3	510
pumsb ($ms_{\%} = 60\%$)	43	15	1.5	10,400
pumsb_star ($ms_{\%} = 50\%$)	58	111	277	2,300
pumsb_star ($ms_{\%} = 40\%$)	**180**	**56**	**310**	**6,700**
retail ($ms_{\%} = 0.1\%$)	42	143	270	143
retail ($ms_{\%} = 0.05\%$)	34	266	720	155
T10I4D100K ($ms_{\%} = 0.5\%$)	47	80	75	1,120
T10I4D100K ($ms_{\%} = 0.1\%$)	60	**320**	380	1,450
T40I10D100K ($ms_{\%} = 10\%$)	7	5	1.3	63
T40I10D100K ($ms_{\%} = 5\%$)	5	4	1.4	9
Average	121	73	780	7,470

Table 6.3: Average speed up of PSM- over benchmark datasets for batch maintenance (when $\Delta^{-} \leq 10\%$).

Algorithm 6 PSM$_\Delta$

Input: $\mathcal{F}_{org}$ the original frequent pattern space represented using equivalence classes; *GE-tree*; ms_{org} and ms_{upd}, the original and updated support threshold.

Output: $\mathcal{F}_{upd\Delta}$ the updated frequent pattern space represented using equivalence classes and the updated *GE-tree*.

Method:

```
 1: $\mathcal{F} := \mathcal{F}_{org}$; {Initialization.}
 2: if $ms_{upd} > ms_{org}$ then
 3:    for all $EC \in \mathcal{F}$ do
 4:       if $EC.support < ms_{upd}$ then
 5:          Remove $EC$ from $\mathcal{F}$;
 6:       end if
 7:    end for
 8: else if $ms_{upd} < ms_{org}$ then
 9:    for all $NG \in$ GE-tree.ngb that $NG.support \geq ms_{upd}$ do
10:       enumNewEC($NG$, $\mathcal{F}$, $ms_{upd}$, GE-tree); {Enumerate new equivalence classes.}
11:    end for
12: end if
13: $\mathcal{F}_{upd\Delta} = \mathcal{F}$
    return $\mathcal{F}_{upd\Delta}$ and the updated GE-tree;
```

6.3 Threshold Adjustment Maintenance: PSM$_\Delta$

This section introduces PSM$_\Delta$, the support threshold adjustment maintenance component of PSM.

Under support threshold adjustment, there are two possible scenarios. When the support threshold is adjusted up, no new frequent patterns and equivalence classes may emerge. Moreover, some existing frequent patterns and equivalence classes may become infrequent. To maintain the pattern space under this scenario, PSM$_\Delta$ basically scans through the existing equivalence classes and remove those that are no longer frequent. On the other hand, when the support threshold is adjusted down, all existing frequent patterns and equivalence classes remain to be frequent. Moreover, new (unknown) frequent patterns and equivalence classes may emerge. The mainte-

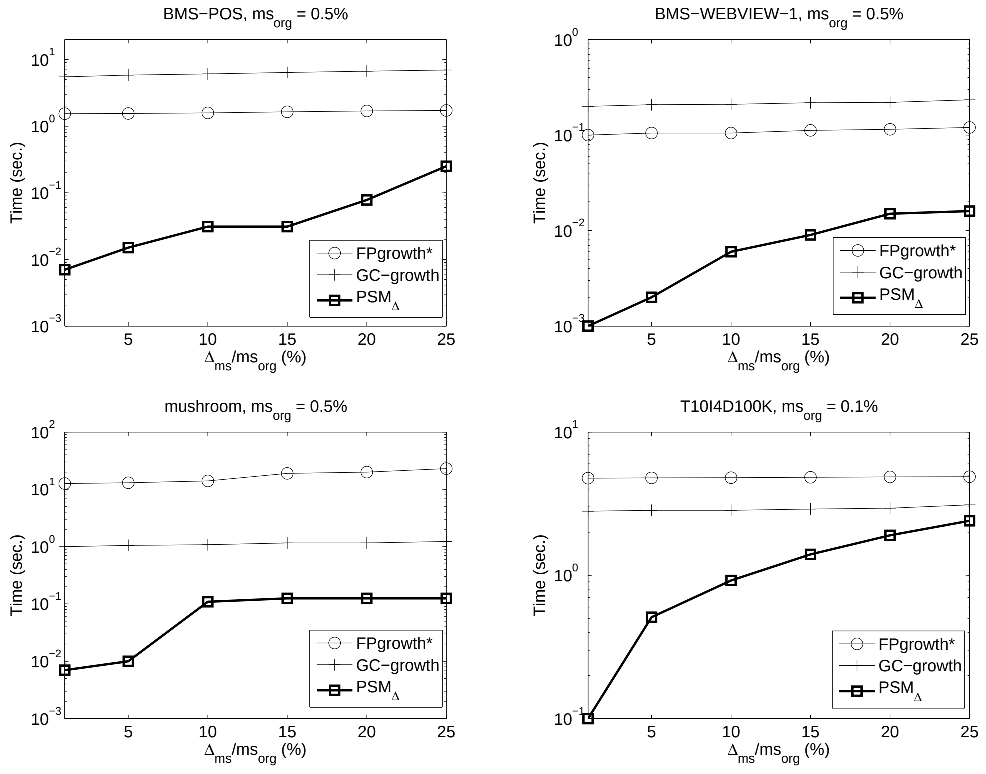


Figure 6.10: Performance comparison of PSM_Δ and the discovery algorithms, FPgrowth* and GC-growth. Notations: Δ_{ms} denotes the difference between the original support threshold and the updated support threshold.

nance for this scenario is much more challenging, for we have little information on the newly emerged patterns. In this case, PSM_Δ efficiently enumerates the newly emerged equivalence classes based on the concepts of *GE-tree* and negative generator border. Details of the enumeration technique can be referred to Section 6.1.2.3 and Procedure 2.

The pseudo-code of PSM_Δ is presented in Algorithm 6, in which the *enumNewEC*() function is defined in Procedure 2. For the case, where the updated threshold is greater than the original one, Line 4 of PSM_Δ filters out all infrequent equivalence classes, and Line 5 then removes them from the pattern space. For the case, where the updated threshold is less than the original one, PSM_Δ first searches

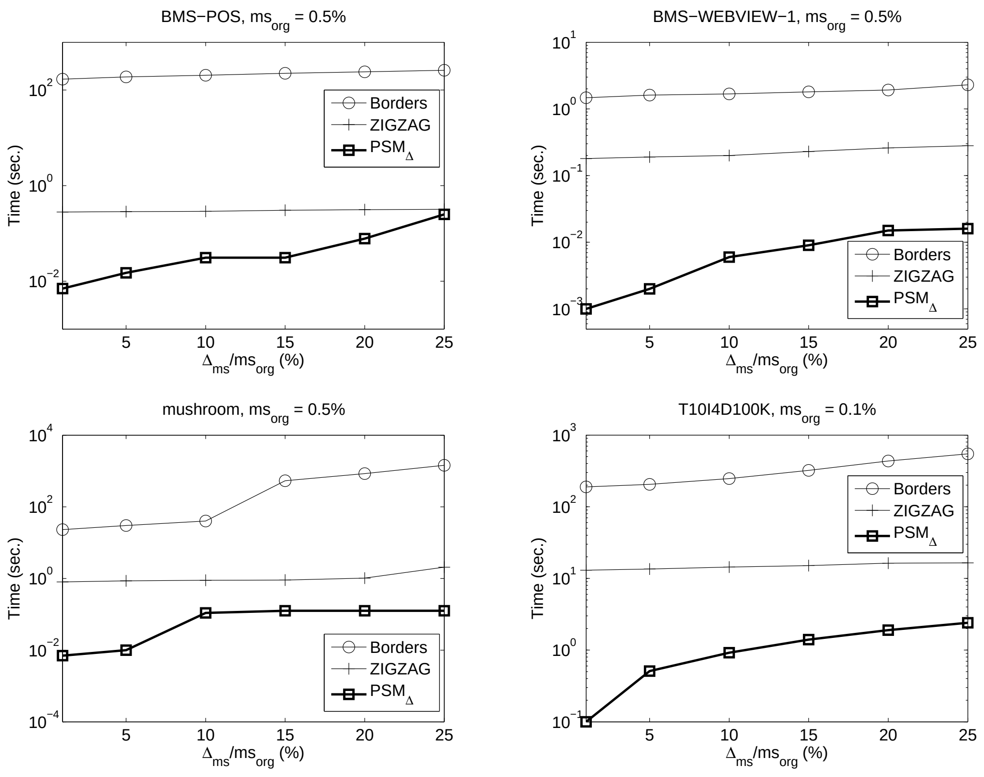


Figure 6.11: Performance comparison of PSM_Δ and the maintenance algorithms, Borders and ZIGZAG. Notations: Δ_{ms} denotes the difference between the original support threshold and the updated support threshold.

for the newly frequent negative border generators (Line 9). We call these generators the "newly promoted generators". PSM_Δ then uses these newly promoted generators as starting points and enumerates the newly frequent equivalence classes (Line 10). As discussed, the enumeration strategy of *GE-tree* ensures the enumeration is correct and complete. As a result, we can conclude that PSM_Δ correctly maintains the frequent pattern space for support threshold adjustments.

We have also evaluated the effectiveness of PSM_Δ through experimental studies. When the threshold is adjusted up, the maintenance process is very straightforward, thus we ignored it in our experimental studies. We focus on the case, where the threshold is adjusted down. PSM_Δ is tested with various degrees of threshold adjustment. Representative experimental results are presented in Figure 6.10 and 6.11. As can been seen from Figure 6.10 and 6.11, PSM_Δ outperforms both the pattern discovery and pattern maintenance algorithms significantly. Moreover, we observe that, for various datasets and initial support thresholds, PSM_Δ is at least twice faster than the discovery algorithms for $\Delta_{ms} \leq 20\%$.

6.4 Summary

This chapter has introduced a novel "maintainer" for the frequent pattern space — Pattern Space Maintainer (PSM). PSM is a "complete maintainer". It address the maintenance of frequent pattern space under incremental updates, decremental updates and support threshold adjustments. PSM is composed of three maintenance components: the incremental component, PSM+, the decremental component, PSM-, and the support threshold adjustment component, PSM_Δ.

The three maintenance components of PSM are developed based on the pattern space evolution analysis. We have investigated the evolution of frequent pattern space under various updates in Chapter 4. In this chapter, we further simplified the evolution scenarios for incremental and decremental updates by considering only one updating transaction at a time. This simplification greatly reduced the complexity of the maintenance problem and allowed more effective algorithms to be developed. Based on the simplified evolution scenarios, we summarized the major computational tasks involved in the maintenance.

We then developed a new data structure, *GE-tree*, to facilitate the maintenance computational tasks. *GE-tree* has many useful features. First, *GE-tree* acts as a compact storage for frequent generators, which are linked with their corresponding equivalence classes. This feature allows the existing equivalence classes to be efficiently updated. Second, *GE-tree*, inspired by *SE-tree*, enumerates new generators in an efficient way that ensures the enumeration is complete and involves no redundancy. Moreover, *GE-tree* stores also the "negative border generators", which conveniently act as starting points to resume the enumeration of new generators. With *GE-tree*, the maintenance components effectively updates the frequent pattern space by updating only the equivalence classes that are affected by the data or threshold updates.

Extensive experiments are conducted to justify the effectiveness of the maintenance components of PSM. PSM is compared with the state-of-the-art frequent pattern discovery and maintenance algorithms. Empirical results have shown that PSM outperforms both the discovery and maintenance algorithms by significant margins.

Part IV

CONCLUSION

Chapter 7

Publications

This chapter lists my publications during my Ph.D. study. Theories and algorithms discussed in this Thesis are derived from these publications. In particular, the survey in Chapter 3 has been published in Publication **III**. Theories discussed in Chapter 4 are extended from Publication **I** and **VI**. Algorithms and some of the results reported in Chapter 5 have been published in Publication **V**. Chapter 6 then combines the results from Publication **II** and **IV**.

Publication List:

I. Mengling Feng, Guozhu Dong, Jinyan Li, Yap-Peng Tan, Limsoon Wong. Pattern space maintenance for data updates & interactive mining. Computational Intelligence, an International Journal, submitted (Invited paper).

II. Mengling Feng, Guozhu Dong, Jinyan Li, Yap-Peng Tan, Limsoon Wong. Evolution of frequent pattern space. Information Processing Letters, submitted.

III. Mengling Feng, Jinyan Li, Yap-Peng Tan, Guozhu Dong, Limsoon Wong. Maintenance of frequent patterns: a survey. (Book Chapter) Post-Mining of Association Rules: Techniques for Effective Knowledge Extraction. Edited by Dr. Yanchang Zhao, Prof. Chengqi Zhang and Dr. Longbing Cao, IGI Global, 2009.

IV. Mengling Feng, Jinyan Li, Yap-Peng Tan, Limsoon Wong. Negative generator border for effective pattern maintenance. International Conference on Advanced Data Mining and Applications 2008: 217-228 (Best Paper Award 1st runner-up).

V. Mengling Feng, Guozhu Dong, Jinyan Li, Yap-Peng Tan, Limsoon Wong. Evolution and maintenance of frequent pattern space when transactions are removed. Pacific-Asia Conference on Knowledge Discovery and Data Mining 2007: 489-497.

VI. Haiquan Li, Jinyan Li, Limsoon Wong, Mengling Feng and Yap-Peng Tan. Relative risk and odds ratio: a data mining perspective. Symposium on Principles of Database Systems 2005: 368-377 (Bi-annual Best Paper Award by I^2R A*STAR).

VII. Mengling Feng and Yap-Peng Tan. Adaptive binarization method for document image analysis. IEEE International Conference on Multimedia & Expo 2004: 339-342.

VIII. Mengling Feng and Yap-Peng Tan. Contrast adaptive binarization of low quality document images. IEICE Electronic Express 1(16): 501- 506, 2004.

Chapter 8

Conclusion

This chapter summarizes the theories and algorithms reported in this Thesis. The future research directions are also identified.

8.1 Summary

Frequent pattern space maintenance is essential for various data mining applications. Pattern maintenance is necessary for database management systems to update the space of frequent patterns in response to various data updates. Pattern maintenance is also an effective solution to answer the hypothetical "what if" queries. In addition, frequent pattern maintenance can also be employed to interactively analyze the development of trends. This Thesis addressed the maintenance of frequent pattern space under three major types of updates: incremental updates, decremental updates and support threshold adjustments.

To better understand the maintenance problem, we conducted a survey on the representative existing maintenance algorithms. The strengths and weaknesses of the existing algorithms have been investigated both algorithmically and experimentally. Through the survey, we observed that most existing algorithms are proposed as an extension of certain pattern discovery algorithms or the data structures they used. But, we believe that, to develop effective maintenance algorithms, it is necessary to understand how the space of frequent patterns evolves under the updates.

We then analyzed how the space of frequent patterns evolve under various types of updates. Since the pattern space is too large to be analyzed directly, we structurally decomposed the pattern space into convex sub-spaces — equivalence classes. We then investigated the evolution of frequent pattern space based on the concept of equivalence classes. The space evolution analysis lays a theoretical foundation for the development of effective maintenance algorithms.

Inspired by the space evolution analysis, we proposed a novel algorithm — "Transaction Removal Update Maintainer" (TRUM) — to maintain the frequent pattern space for decremental updates. A new data structure, *Transaction-ID Tree* (*TID-tree*), has also been developed to facilitate the computational tasks involved in TRUM. With *TID-tree*, TRUM effectively maintains the pattern space by updating only the equivalence classes that are affected by the decremental update. The effectiveness of TRUM was justified through experimental evaluations.

Since TRUM only addresses the decremental maintenance, we further proposed a "complete maintainer" for the space of frequent patterns. The maintainer is called the "Pattern Space Maintainer", in short PSM. It is called the "complete maintainer" for it addresses three different types updates: incremental updates, decremental updates and support threshold adjustments. PSM, thus, is composed of three maintenance components: the incremental component, PSM+, the decremental component, PSM-, and the support threshold adjustment component, PSM_{Δ}. The three maintenance components of PSM are also developed based on the pattern space evolution analysis. Inspired by the evolution analysis, we introduced a new maintenance data structure, *Generator-Enumeration Tree* (*GE-tree*). *GE-tree* effectively assists both the update of existing equivalence classes and the generation newly emerged classes. Our experimental studies have shown that PSM outperforms the state-of-the-art discovery and maintenance algorithms by significant margins.

8.2 Future Directions

Three avenues of future research are presented here. The first is to apply our proposed "maintainers", TRUM and PSM, to discover and analyze trends in real applications. The second is to explore the maintenance of frequent patterns over data steams. The third is to generalize our maintenance framework to other types of patterns.

The first avenue is to apply our proposed maintenance methods to solve real problems in trend analysis applications. Trends that are always around are usually neither useful nor interesting. On the other hand, newly emerged trends and trends that just disappeared are more interesting and meaningful ones. When a new trend has emerged or an existing trend has vanished, TRUM can be used to analyze the trend development in a retrospective manner. For PSM, since it addresses both incremental and decremental maintenance, it can be employed to analyze the trend in both retrospective and prospective manners. With the proposed maintenance methods, we can effectively find out exactly when the emergence or disappearance of the trend happens, which allows us to further investigate the causes behind the trend development.

The second avenue is to explore the maintenance of frequent patterns over "data streams". A "data stream" is an ordered sequence of transactions that arrives in timely order. Data streams are involved in many applications, e.g. sensor network monitoring [29], internet packet frequency estimation [19], web failure analysis [11], etc. Compared with the conventional transaction dataset, the frequent pattern maintenance in data streams is more challenging due to the following factors. First, data streams are continuous and unbounded [42]. While handling data streams, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Second, data in streams are not necessarily uniformly distributed [42]. That is to say patterns that are currently infrequent may emerge to be frequent in the future, and vice versa. Therefore, we can no longer simply prune out infrequent patterns. Third, data streams can be collected in various manners, e.g. the "sliding widow" manner and the "damped" manner [42]. Therefore, to handle frequent pattern space maintenance for data streams, an efficient and versatile algorithm is needed.

The third avenue is to generalize our maintenance framework to address the maintenance of other types of patterns, e.g. conditional functional dependent rules [22], matching rules [23], odds ratio patterns [47], emerging patterns [21] and discriminative emerging patterns [48]. The migration of our maintenance framework to other types of pattern space is not an easy task. Unlike frequent patterns, many types of patterns in data mining do not enjoy the *a priori* property, and, more importantly, their pattern space is not convex. For those non-convex pattern spaces, our maintenance method cannot be directly applied. Possible solutions to this problem are: (1) decompose the pattern space into sub-spaces that are convex, or (2) approximate the pattern space to a convex space.

Bibliography

[1] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] Rakesh Agrawal and Tomasz Imielinski. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14, 1995.

[5] Yonatan Aumann, Ronen Feldman, Orly Lipshtat, and Heikki Manilla. Borders: an efficient algorithm for association generation in dynamic databases. *Journal of Intelligent Information System*, 12(1):61–73, 1999.

[6] Necip Fazil Ayan, Abdullah Uz Tansel, and M. Erol Arkun. An efficient algorithm to update large itemsets with early pruning. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 287–291, 1999.

[7] Jean-François Boulicaut, Artur Bykowski, and Christophe Rigotti. Free-sets: a condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery*, 7(1):5–22, 2003.

[8] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: generalizing association rules to correlations. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 265–276, 1997.

[9] Douglas Burdick, Manuel Calimlim, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Mafia: a maximal frequent itemset algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 17(11):1490–1504, 2005.

[10] Artur Bykowski and Christophe Rigotti. A condensed representation to find frequent patterns. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 267–273, 2001.

[11] Y. Dora Cai, David Clutter, Greg Pape, Jiawei Han, Michael Welge, and Loretta Auvil. Maids: mining alarming incidents from data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 919–920, 2004.

[12] Chia-Hui Chang and Shi-Hsan Yang. Enhancing SWF for incremental association mining by itemset maintenance. In *Proceedings of 7th Pacific-Asia Con-*

ference on Advances in Knowledge Discovery and Data Mining, pages 301–312, 2003.

[13] Lee Chao. *Database Development and Management (Foundations of Database Design)*. Auerbach Publications, 2006.

[14] David Wai-Lok Cheung, Jiawei Han, Vincent T. Y. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: an incremental updating technique. In *Proceedings of the 12th International Conference On Data Engineering*, pages 106–114, 1996.

[15] David Wai-Lok Cheung, Sau Dan Lee, and Ben Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, pages 185–194, 1997.

[16] William Cheung and Osmar R. Zaïane. Incremental mining of frequent patterns without candidate generation or support constraint. In *Proceedings of 7th International Database Engineering and Applications Symposium*, pages 111–116, 2003.

[17] Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Muntz. Moment: maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the 4th IEEE International Conference on Data Mining*, pages 59–66, 2004.

[18] Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Muntz. Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. *Knowledge and Information Systems*, 10(3):265–294, 2006.

[19] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of 10th Annual European Symposium on Algorithms*, pages 348–360, 2002.

[20] Guozhu Dong, Chunyu Jiang, Jian Pei, Jinyan Li, and Limsoon Wong. Mining succinct systems of minimal generators of formal concepts. In *Proceedings of 10th International Conference on Database Systems for Advanced Applications*, pages 175–187, 2005.

[21] Guozhu Dong and Jinyan Li. Efficient mining of emerging patterns: discovering trends and differences. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 43–52, 1999.

[22] Wenfei Fan, Floris Geerts, Laks V. S. Lakshmanan, and Ming Xiong. Discovering conditional functional dependencies. In *Proceedings of the 25th International Conference on Data Engineering*, pages 1231–1234, 2009.

[23] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *Proceedings of the VLDB Endowment*, 2(1):407–418, 2009.

[24] Bart Goethals. http://www.adrem.ua.ac.be/~goethals/software/.

[25] Bart Goethals and Mohammed Javeed Zaki, editors. *FIMI '03, Frequent Itemset Mining Implementations*, 2003.

[26] Karam Gouda and Mohammed Javeed Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 163–170, 2001.

[27] Karam Gouda and Mohammed Javeed Zaki. Genmax: an efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3):223–242, 2005.

[28] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.

[29] Mihail Halatchev and Le Gruenwald. Estimating missing values in related sensor data streams. In *Proceedings of the 11th International Conference on Management of Data*, pages 83–94, 2005.

[30] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of the 15th International Conference on Data Engineering*, pages 106–115, 1999.

[31] Jiawei Han and Yongjian Fu. Discovery of multiple-level association rules from large databases. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of 21th International Conference on Very Large Data Bases*, pages 420–431, 1995.

[32] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques.* Morgan Kaufmann Publishers, 2001.

[33] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2000.

[34] Hao Huang, Xindong Wu, and Richard Relue. Association analysis with one scan of databases. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, pages 629–632, 2002.

[35] Roberto J. Bayardo Jr. Efficiently mining long patterns from databases. In *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data*, pages 85–93, 1998.

[36] Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, Hannu Toivonen, and A. Inkeri Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proceedings of the 3rd International Conference on Information and Knowledge Management*, pages 401–407, 1994.

[37] Jia-Ling Koh and Shui-Feng Shieh. An efficient approach for maintaining association rules based on adjusting fp-tree structures. In *Proceedings of the 9th International Conference on Database Systems for Advanced Applications*, pages 417–424, 2004.

[38] Marzena Kryszkiewicz and Marcin Gajek. Concise representation of frequent patterns based on generalized disjunction-free generators. In *Proceedings of 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, pages 159–171, 2002.

[39] Chang-Hung Lee, Cheng-Ru Lin, and Ming-Syan Chen. Sliding window filtering: an efficient method for incremental mining on a time-variant database. *Information Systems*, 30(3):227–244, 2005.

[40] Sau Dan Lee, David Wai-Lok Cheung, and Ben Kao. Is sampling useful in data mining? a case in the maintenance of discovered association rules. *Data Mining and Knowledge Discovery*, 2(3):233–262, 1998.

[41] Brian Lent, Arun Swami, and Jennifer Widom. Clustering association rules. In *Proceedings of 13th International Conference on Data Engineering*, pages 220–231, 1997.

[42] Carson Kai-Sang Leung and Quamrul I. Khan. Dstree: a tree structure for the mining of frequent sets from data streams. In *Proceedings of the 6th IEEE International Conference on Data Mining*, pages 928–932, 2006.

[43] Carson Kai-Sang Leung, Quamrul I. Khan, and Tariqul Hoque. Cantree: a tree structure for efficient incremental mining of frequent patterns. In *Proceedings of the 5th IEEE International Conference on Data Mining*, pages 274–281, 2005.

[44] Carson Kai-Sang Leung, Quamrul I. Khan, Zhan Li, and Tariqul Hoque. Cantree: a canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311, 2007.

[45] Cuiping Li, Gao Cong, Anthony K. H. Tung, and Shan Wang. Incremental maintenance of quotient cube for median. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 226–235, 2004.

[46] Cuiping Li, Kum-Hoe Tung, and Shan Wang. Incremental maintenance of quotient cube based on galois lattice. *Journal of Computer Science and Technology*, 19(3):302–308, 2004.

[47] Haiquan Li, Jinyan Li, Limsoon Wong, Mengling Feng, and Yap-Peng Tan. Relative risk and odds ratio: a data mining perspective. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 368–377, 2005.

[48] Jinyan Li, Guimei Liu, and Limsoon Wong. Mining statistically important equivalence classes and delta-discriminative emerging patterns. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 430–439, 2007.

[49] Guimei Liu, Jinyan Li, and Limsoon Wong. A new concise representation of frequent itemsets using generators and a positive border. *Knowledge and Information Systems*, 17(1):35–56, 2008.

[50] Junqiang Liu, Yunhe Pan, Ke Wang, and Jiawei Han. Mining frequent item sets by opportunistic projection. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 229–238, 2002.

[51] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[52] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *Proceedings of AAAI'94 Workshop Knowldege Discovery in Databases*, pages 181–192, 1994.

[53] H. D. K. Moonesinghe, Samah Jamal Fodeh, and Pang-Ning Tan. Frequent closed itemset mining using prefix graphs with an efficient flow-based pruning strategy. In *Proceedings of the 6th IEEE International Conference on Data Mining*, pages 426–435, 2006.

[54] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 13–24, 1998.

[55] Edward Omiecinski and Ashok Savasere. Efficient mining of association rules in large dynamic databases. In *Proceedings of 16th British National Conferenc on Databases Advances in Databases*, pages 49–63, 1998.

[56] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, 1995.

[57] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of 7th International Conference on Database Theory*, pages 398–416, 1999.

[58] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999.

[59] 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, pages 441–448, 2001.

[60] Jian Pei, Jiawei Han, and Runying Mao. Closet: an efficient algorithm for mining frequent closed itemsets. In *Proceedings of 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[61] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2002.

[62] G. Ramalingam. *Bounded Incremental Computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer, 1996.

[63] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating mining with relational database systems: alternatives and implications. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 343–354, 1998.

[64] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of 21th International Conference on Very Large Data Bases*, pages 432–444, 1995.

[65] Craig Silverstein, Sergey Brin, Rajeev Motwani, and Jeffrey D. Ullman. Scalable techniques for mining causal structures. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pages 594–605, 1998.

[66] Mingjun Song and Sanguthevar Rajasekaran. Finding frequent itemsets by transaction mapping. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 488–492, 2005.

[67] Mingjun Song and Sanguthevar Rajasekaran. A transaction mapping algorithm for frequent itemsets mining. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):472–481, 2006.

[68] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[69] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. Lcm: an efficient algorithm for enumerating frequent closed item sets. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.

[70] Adriano Veloso, Wagner Meira Jr., Márcio de Carvalho, Bruno Pôssas, Srinivasan Parthasarathy, and Mohammed Javeed Zaki. Mining frequent itemsets in evolving databases. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, 2002.

[71] Chao Wang and Srinivasan Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 31–40, 2004.

[72] Jianyong Wang, Jiawei Han, and Jian Pei. Closet+: searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 236–245, 2003.

[73] Ke Wang, Yu He, and Jiawei Han. Mining frequent itemsets using support constraints. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 43–52, 2000.

[74] Mohammed Javeed Zaki and Ching-Jiu Hsiao. Charm: an efficient algorithm for closed itemset mining. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, 2002.

[75] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–286, 1997.

[76] Shichao Zhang, Xindong Wu, Jilian Zhang, and Chengqi Zhang. A decremental algorithm for maintaining frequent itemsets in dynamic databases. In *Proceedings of 7th International Conference on Data Warehousing and Knowledge Discovery*, pages 305–314, 2005.

[77] Zequn Zhou and C. I. Ezeife. A low-scan incremental association rule maintenance method based on the apriori property. In *Proceedings of 14th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 26–35, 2001.