Pattern Space Maintenance for Data Updates and Interactive Mining *

Mengling Feng †[1,3,4] Guozhu Dong,[2] Jinyan Li,[1]
Yap-Peng Tan,[1] Limsoon Wong [3]
[1] Nanyang Technological University,[2]Wright State University
[3]National University of Singapore, [4]Institute for Infocomm Research

Abstract

This paper addresses the incremental and decremental maintenance of the frequent pattern space. We conduct an in-depth investigation on how the frequent pattern space evolves under both incremental and decremental updates. Based on the evolution analysis, a new data structure, *Generator-Enumeration Tree* (*GE-tree*), is developed to facilitate the maintenance of the frequent pattern space. With the concept of *GE-tree*, we propose two novel algorithms, *Pattern Space Maintainer+* (PSM+) and *Pattern Space Maintainer-* (PSM-), for the incremental and decremental maintenance of frequent patterns. Experimental results demonstrate that the proposed algorithms, on average, outperform the representative state-of-the-art methods by an order of magnitude.

Key words: Data Mining, Frequent Pattern, Incremental Maintenance, Data Update & Interactive Mining.

*This manuscript is to be submitted to the Special Issue of Computational Intelligence on "Advanced Data Mining and Applications". The conference version of this paper was published in ADMA'2008. Editors: Charles Ling and Qiang Yang (Guest Editors).

†Corresponding author. Email: mornin@gmail.com

1 Introduction

Updates are a fundamental aspect of data management. Updates allow obsolete and incorrect records to be removed and new records to be included. When a database is updated frequently, repeating the pattern discovery process from scratch during each update causes significant computational and I/O overheads. Therefore, it is important to analyze how the discovered patterns may change in response to updates, and to formulate more effective algorithms to maintain the discovered patterns on the updated database.

Pattern maintenance is also useful for interactive mining applications. For example, pattern maintenance can be used to interactively analyze the evolution trend of a time series data. This type of trend analysis usually focuses on a certain period of time, and patterns before the targeted period are first extracted as a reference. Then records within the targeted period are inserted one by one in time sequence. The patterns before and after the insertion are then compared to find whether new patterns (trends) have emerged and how the existing patterns (trends) have changed. Such an interactive study is a useful tool to detect significant events, like the emergence of new trend, changes of the existing trends, vanishing trends, etc. More importantly, through the study, we can also identify the time when the significant events happened, which allows further investigation on the causes of the events. This type of "before vs. after" analysis requires intensive pattern discovery and comparison computation. Solving the problem using conventional pattern discovery methods involves large amount of redundancies, and pattern maintenance can be used to effectively avoid these redundancies.

This paper addresses the maintenance of the frequent patterns space. Frequent patterns (Agrawal and Imielinski, 1993) are a very important type of patterns in data mining. Frequent patterns play an essential role in various

knowledge discovery tasks, such as the discovery of association rules, correlations, causality, sequential patterns, emerging patterns, etc. The frequent patterns space, consisting of all the frequent patterns, is usually very large. Thus, the maintenance of the frequent pattern space is computationally challenging.

We focus on two major types of updates in data management and interactive mining. The first type, where new transactions are inserted into the original dataset, is called an *incremental update*. The associated maintenance process is called *incremental maintenance*. The second type, where some transactions are removed from the original dataset, is called a *decremental update*. The associated maintenance process is called *decremental maintenance*.

Our contributions in this paper are as follows. (1) We analyze how the space of frequent patterns evolves under both incremental and decremental updates. The space of frequent patterns is too huge to be studied directly. Therefore, we propose to structurally decompose the pattern space into subspaces — equivalence classes. This structural decomposition of the frequent pattern space allows us to concisely represent the space with the borders of equivalence classes; the decomposition also makes it possible to formally describe the evolution of the pattern space based on the changes of equivalence classes; and, more importantly, the decomposition enables us to maintain the frequent pattern space in a divide-and-conquer manner. (2) Based on the space evolution analysis, we summarize the major computation tasks involved in frequent pattern maintenance. (3) To effectively perform the maintenance tasks, we develop a data structure, *Generator-Enumeration Tree* (*GE-tree*). *GE-tree* helps us efficiently locate and update equivalence classes that are affected by the updates, and it also ensures complete enumeration of new equivalence classes without any redundancy. (4) We propose two novel maintenance algorithms, *Pattern Space Maintainer+* (PSM+) and *Pattern Space Maintainer-* (PSM-). With *GE-tree*,

PSM+ and PSM- effectively maintain the frequent pattern space for incremental and decremental updates. PSM+ and PSM- can be easily integrated, and we name the integrated maintainer the *Pattern Space Maintainer* (PSM). We also demonstrate that PSM can be extended to update the frequent pattern space for support threshold adjustments. (5) We have conducted extensive experiments to evaluate the effectiveness of our proposed algorithms. Experimental results show that the proposed algorithms, on average, outperform the state-of-the-art approaches by more than an order of magnitude.

The rest of the paper is organized as follows. In Section 2, we review the related works of frequent pattern maintenance. In Section 3, we formally define the maintenance problem. In Section 4, we investigate how the space of frequent pattern can be structurally decomposed into and represented by equivalence classes. In Section 5 and 6, we discuss the proposed incremental and decremental maintenance algorithms. The generalization and extension of the proposed algorithms are discussed in Section 7, and the experimental results are presented in Section 8. We conclude the paper in Section 9.

2 Related Work

In the literature, the frequent pattern maintenance algorithms can be classified into four main categories: the 1) *Apriori-based* algorithms, 2) *Partition-based* algorithms, 3) *Prefix-tree-based* algorithms and 4) *Concise-representation-based* algorithms.

FUP (Cheung *et al.*, 1996) is the first *Apriori*-based maintenance algorithm. FUP focuses on the incremental maintenance of frequent patterns. Inspired by Apriori (Agrawal and Imielinski, 1993), FUP updates the space of frequent patterns iteratively based on the candidate-generation-verification framework. The key technique of FUP is to make use of support information in previously dis-

covered frequent patterns to reduce the number of candidate patterns. Since the performance of candidate-generation-verification based algorithms heavily depends on the size of the candidate set, FUP outperforms Apriori. FUP is then generalized as FUP2H (Cheung *et al.*, 1997) to handle both incremental and decremental maintenance. Similarly, the partition-based algorithm SWF (Lee *et al.*, 2005) also employs the candidate-generation-verification framework. However, SWF applies different techniques to reduce the size of candidate set. SWF slices a dataset into several partitions and employs a filtering threshold in each partition to filter out unnecessary candidate patterns. Even with all the candidate reduction techniques, the candidate-generation-verification framework still leads to the enumeration of large number of unnecessary candidates. This greatly limits the performance of both *Apriori*-based and partition-based algorithms.

To address this shortcoming of the candidate-generation-verification framework, prefix-tree-based algorithms, such as CanTree (Leung *et al.*, 2007), that involve no candidate generation are proposed. CanTree evolves from FP-growth (Han *et al.*, 2000) — the state-of-the-art prefix-tree-based frequent pattern discovery algorithm. CanTree arranges items according to some fixed canonical order that will not be affected by data updates. This allows new transactions to be efficiently inserted into the existing prefix-tree without node swapping/merging. However, prefix-tree based algorithms still suffer from the undesirably large size of the frequent pattern space.

To break this bottleneck, concise representations of the frequent pattern space are proposed. The commonly used representations include "maximal patterns" (Bayardo, 1998), "closed patterns" and "generators" (Pasquier *et al.*, 1999). Algorithms have also been proposed to maintain the concise representations. Moment (Chi *et al.*, 2006) is one example. Moment dynamically maintains

the frequent closed patterns. Moment focuses on a special update scenario where each time only one new transaction is inserted and one obsolete transaction is removed, and thus it is proposed based on the hypothesis that there are only *small changes* to the frequent closed patterns given a small amount of updates. Due to this unfavorable constraint, the performance of Moment degrades dramatically when the number of updates gets large. ZIGZAG (Veloso *et al.*, 2002), on the other hand, maintains the maximal patterns. Extended from the maximal pattern discovery algorithm GENMAX (Gouda and Zaki, 2001), ZIGZAG updates the maximal patterns by a backtracking search, which is guided by the outcomes of the previous maintenance iteration. However, the maximal patterns are a lossy representation of the frequent pattern space, which do not provide support information of frequent patterns.

We observe that most of the prior works in frequent pattern maintenance, e.g. FUP, CanTree and ZIGZAG, are proposed as an extension of frequent pattern discovery algorithms. Unlike these prior works, we propose our maintenance algorithms based on an in-depth analysis on the evolution of the pattern space under data updates. The evolution of the pattern space is analyzed using the concept of equivalence classes. Different from the maximal pattern in ZIGZAG, the equivalence class is a lossless [1] concise representation of the frequent pattern space. Also, unlike Moment, which bears some unfavorable assumptions, our maintenance algorithms aim to handle batch updates.

[1] We say a representation is lossless if it is sufficient to derive and determine the support of all frequent patterns without accessing the datasets.

3 Problem Definition

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 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 $ms_a = \lceil ms_{\%} \times |\mathcal{D}| \rceil$. For this paper, we assume the percentage support threshold is used unless otherwise specified. Given $ms_{\%}$ or ms_a, a pattern P is said to be *frequent* in a dataset $\mathcal{D}$ iff $sup(P, \mathcal{D}) \geq ms_a = \lceil ms_{\%} \times |\mathcal{D}| \rceil$. 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 $\mathcal{F}(\mathcal{D}, ms_a)$.

For incremental maintenance, we use the following notations: $\mathcal{D}_{org}$ is the original dataset, $\mathcal{D}_{inc}$ — the incremental dataset — 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 $\mathcal{D}_{org} \cap \mathcal{D}_{inc} = \emptyset$. 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 $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$.

Analogously, we use the following notations for decremental maintenance: $\mathcal{D}_{dec}$ — the decremental dataset — 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 $\mathcal{D}_{dec} \subseteq \mathcal{D}_{org}$. Thus $|\mathcal{D}_{upd-}| = |\mathcal{D}_{org}| - |\mathcal{D}_{dec}|$. Given $ms_{\%}$, the task of decremental maintenance is to obtain the updated frequent pattern

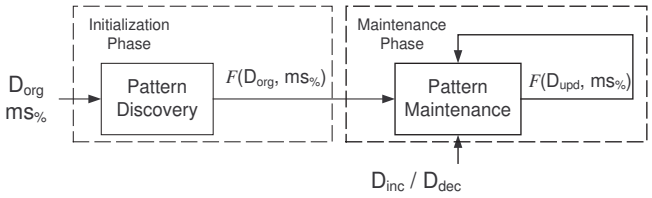

Figure 1: Process of pattern maintenance. Notations: $\mathcal{D}_{org}$, $\mathcal{D}_{inc}$, $\mathcal{D}_{dec}$ and $\mathcal{D}_{upd}$ denote the original, the incremental, the decremental and the updated datasets respectively; $ms_{\%}$ is the minimum support threshold; and $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$ and $\mathcal{F}(\mathcal{D}_{upd}, ms_{\%})$ refer to the original and updated frequent pattern space.

space $\mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$ by updating the original pattern space $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$.

In pattern maintenance applications, the maintenance process, as illustrated in Figure 1, consists of two phases: the initialization phase and the maintenance phase. Given the original dataset, $\mathcal{D}_{org}$, and the minimum support threshold, $ms_{\%}$, a pattern discovery algorithm is employed in the initialization phase to generate the original frequent pattern space $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$. In this paper, the discovery algorithm GC-growth (Li *et al.*, 2005) is used. Note that the discovery of the original pattern space needs to be done only once as initialization for the subsequent updates. Therefore, the initialization phase is not considered while evaluating the performance of maintenance algorithms. The initialization phase is not our focus. We focus on the maintenance phase. In the maintenance phase, a maintenance algorithm is employed. The maintenance algorithm takes the original pattern space $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$ as input and updates the space based on the data updates, $\mathcal{D}_{inc}/\mathcal{D}_{dec}$. The updated pattern space $\mathcal{F}(\mathcal{D}_{upd}, ms_{\%})$ is then input back into the maintenance algorithm for subsequent updates. The objective of this paper is to develop an efficient maintenance algorithm for the frequent pattern space.

4 Structural Decomposition of Pattern Space

Understanding how the frequent pattern space evolves when data is updated is essential for effective maintenance of the space. 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.

We observe that the frequent pattern space is a convex space.

Definition 4.1 (Convex Space) *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 a "bound" of $\mathcal{S}$. 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 another bound of $\mathcal{S}$. A pattern X is most specific in $\mathcal{S}$ if there is no proper superset of X in $\mathcal{S}$. We call the former bound the "left bound" of $\mathcal{S}$, denoted $\mathcal{L}$; and the latter bound the "right bound" of $\mathcal{S}$, denoted $\mathcal{R}$. We call the pair of left and right bound the "border" of $\mathcal{S}$, which is denoted by $\langle \mathcal{L}, \mathcal{R} \rangle$. It is easy to show that a convex space can be concisely represented by its borders without loss of information.

Fact 4.2 (Cf. Li *et al.* (2005)) *$\mathcal{F}(ms_{\%}, \mathcal{D})$ is convex. Furthermore, it can be structurally decomposed into convex sub-spaces — equivalence classes.*

We further found that, due to its convexity, the frequent pattern space can be structurally decomposed into sub-spaces, which are much smaller in terms of size. The sub-space is called the equivalence class, and it is formally defined as follows.

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* $[P]_{\mathcal{D}} = \{Q \mid f(P, \mathcal{D}) = f(Q, \mathcal{D})$*,* Q *is a pattern in* $\mathcal{D}\}$.

In other words, 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 a given equivalence class have the same support. So we extend the notations and write $sup(P, \mathcal{D})$ to denote the support of an equivalence class $[P]_{\mathcal{D}}$ and $P \in \mathcal{F}(ms, \mathcal{D})$ to mean the equivalence class is frequent. Furthermore, equivalence classes are also convex and thus they can be compactly represented by their borders without loss of information (Li *et al.*, 2005). The right bound of an equivalence class is actually a closed pattern, and the left bound is a group of generators (key patterns).

Definition 4.4 (Generator & Closed Pattern (Pasquier *et al.*, 1999)) *A pattern* P *is a "key pattern" or a "generator" in a dataset* $\mathcal{D}$ *iff for every* $P' \subset P$*, it is the case that* $sup(P', \mathcal{D}) > sup(P, \mathcal{D})$*. In contrast, a pattern* P *is a "closed pattern" in a dataset* $\mathcal{D}$ *iff for every* $P' \supset P$*, it is the case that* $sup(P', \mathcal{D}) < sup(P, \mathcal{D})$.

Based on the definition of the border of a convex space, we can define generators and closed patterns in an alternative way.

Fact 4.5 *A pattern* P *is a key pattern or a generator 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* $[P]_{\mathcal{D}}$.

Therefore, the closed pattern and generators form the border of the corresponding equivalence class, and they, furthermore, 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.

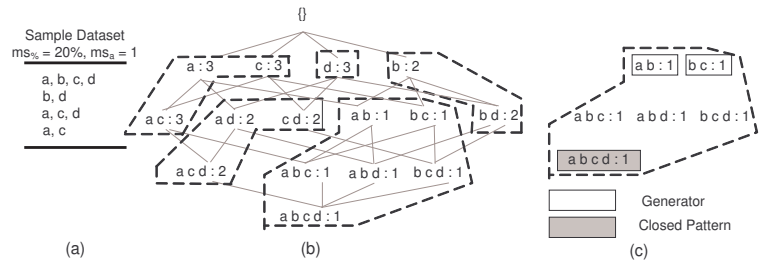

Figure 2: Demonstration of the structural decomposition of the frequent pattern space. (a)The sample dataset; (b) decomposition of the frequent pattern space of the sample dataset into 5 equivalence classes; (c) the "border" of an equivalence class.

Figure 2 (b) shows the frequent pattern space for the sample dataset in (a) when $ms_{\%} = 20\%/ms_a = 1$. Figure 2 (b) also graphically demonstrates how the pattern space, which consists of 15 patterns, can be structurally decomposed into 5 equivalence classes. Figure 2 (c) then demonstrates how an equivalence class can be concisely represented by its border patterns — the generators and closed pattern.

In addition, we observe that generators follow the "a priori" (or anti-monotone) property.

Fact 4.6 (Cf. Li *et al.* (2005)) *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, then every subset of P is also a generator in $\mathcal{D}$. Thus, if P is a frequent generator, then every subset of P is also a frequent generator in $\mathcal{D}$.*

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 (Li *et al.*, 2004). 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.

5 Incremental Maintenance of Pattern Space

This section discusses the incremental maintenance of the frequent pattern space. 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 $ms_{\%}$, the task of incremental maintenance is to obtain the updated pattern space by maintaining the original pattern space.

To develop effective incremental maintenance algorithm, we start off with a study on the evolution of the frequent pattern space under incremental updates using the concept of equivalence class. Through the space evolution study, we summarize the major computational tasks in the incremental maintenance. To complete the computational tasks efficiently, we develop a new data structure, *Generator-Enumeration Tree* (*GE-tree*). Based on the *GE-tree*, a novel incremental maintenance algorithm, named *Pattern Space Maintainer+* (PSM+), is proposed.

5.1 Evolution of Pattern Space

We first investigate how the existing (frequent) equivalence classes evolve when new transactions are added. We observe that, after an incremental update, the support of an equivalence class can only increase and the size of an equivalence class can only shrink.

Proposition 5.1 *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 $\mathcal{D}_{inc} \cap \mathcal{D}_{org} = \emptyset$, we have $f(Q, \mathcal{D}_{org}) = f(P, \mathcal{D}_{org})$. This means $Q \in [P]_{\mathcal{D}_{org}}$ for every $Q \in [P]_{\mathcal{D}_{upd+}}$. Thus we can conclude $[P]_{\mathcal{D}_{upd+}} \subseteq [P]_{\mathcal{D}_{org}}$. Also, $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{inc}) \geq sup(P, \mathcal{D}_{org})$.* □

In particular, we discover that, under an incremental update, the existing equivalence classes evolve in three different ways. The first way is to remain unchanged without any change in support, such as $EC2$ in Figure 3 (a). The second way is to remain unchanged but with an increased support, such as $EC3$ and $EC4$ in Figure 3 (a). The third way is to split into two or more classes, such as $EC1$ in Figure 3 (a). In this case, the size of equivalence classes will shrink as described in Proposition 5.1. On the other hand, an incremental update may induce new [2] (frequent) equivalence classes to emerge. E.g. $EC5'$ in Figure 3 (a).

To have an in-depth understanding on how the pattern space evolve under the incremental update, we now investigate the exact conditions for the three ways that existing equivalence classes may evolve and also the conditions for new equivalence classes to emerge. We denote the closed pattern of an equivalence class $[p]_{\mathcal{D}}$ as $Clo([p]_{\mathcal{D}})$ and the generators or key patterns of $[p]_{\mathcal{D}}$ as $Keys([p]_{\mathcal{D}})$. We assume the incremental dataset $\mathcal{D}_{inc}$ contains only one transaction t_+ for ease of discussion.

Theorem 5.2 *Let $\mathcal{D}_{org}$ be the original dataset, $\mathcal{D}_{inc}$ be the incremental dataset, $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \mathcal{D}_{inc}$ and $ms_{\%}$ be the support threshold. Suppose $\mathcal{D}_{inc}$ consists*

[2] We call an equivalence class "new" iff the patterns in the class are not in the original pattern space but in the updated pattern space.

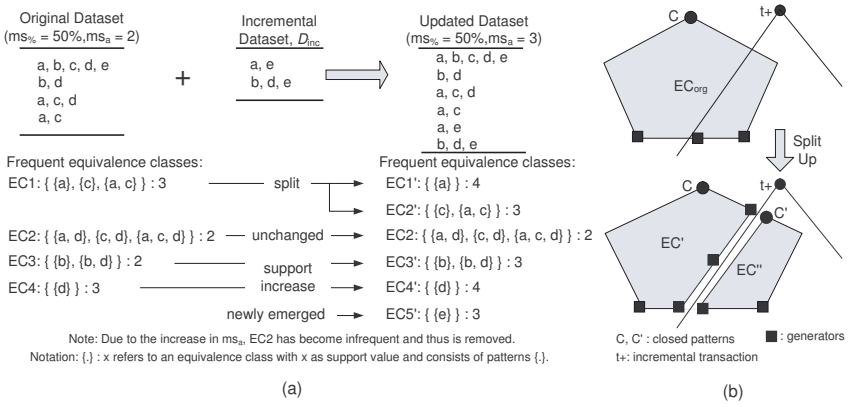

Figure 3: (a) The evolution of the frequent pattern space under the incremental update; (b) the splitting up of an equivalence class EC_{org} after $t+$ is inserted.

of only one transaction t_+. *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* $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + 1$.

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.* $[P]_{\mathcal{D}_{upd+}} = \{Q | Q \in [P]_{\mathcal{D}_{org}} \wedge Q \subseteq t_+\}$, $Clo([P]_{\mathcal{D}_{upd+}}) = Clo([P]_{\mathcal{D}_{org}}) \cap t_+$ *and* $Keys([P]_{\mathcal{D}_{upd+}}) = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$.

4. $P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, $P \not\subseteq t_+$ *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,* $[P]_{\mathcal{D}_{upd+}} = \{Q|Q \in [P]_{\mathcal{D}_{org}} \wedge Q \not\subseteq t_+\}$, $Clo([P]_{\mathcal{D}_{upd+}}) = Clo([P]_{\mathcal{D}_{org}})$ *and* $Keys([P]_{\mathcal{D}_{upd+}}) = min\{\{K|K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \not\subseteq t_+\} \cup \{K' \cup \{x_i\}, i = 1, 2, \cdots | K' \in Keys([P]_{\mathcal{D}_{org}}) \wedge K' \subseteq t_+, x_i \in Clo([P]_{\mathcal{D}_{org}}) \wedge x_i \notin t_+\}\}$.

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* $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + 1$.

Proof: *Refer to Appendix.* □

Scenario 3 and 4 in Theorem 5.2 describe the cases where an existing equivalence class splits. The splitting up of an equivalence class is a bit complicated. Thus a graphical example is shown in Figure 3 (b). The original equivalence class EC_{org} splits up due to the insertion of transaction t_+. The resulting equivalence class EC'' corresponds to the equivalence class $[P]_{\mathcal{D}_{upd+}}$ described in Scenario 3, and EC' corresponds to $[P]_{\mathcal{D}_{upd+}}$ described in Scenario 4.

Theorem 5.2 summarizes how the frequent pattern space evolves when a new transaction is inserted. More importantly, the theorem describes how the updated frequent equivalence classes of $\mathcal{D}_{upd+}$ can be derived from the existing frequent equivalence classes of $\mathcal{D}_{org}$. Theorem 5.2 provides us a theoretical framework for effective incremental maintenance of the frequent pattern space. Note that: although the theorem focuses on the case where only one new transaction is inserted, it is also applicable to batch updates [3]. Suppose

[3] A generalized version of Theorem 5.2, which describes how the frequent pattern space evolves when a batch of new transactions are added, is presented in Feng *et al.* (2009).

$\mathcal{D}_{inc} = \{t_1, \cdots, t_n\}$. To obtain the updated pattern space $\mathcal{F}(\mathcal{D}_{upd+}, ms_{\%})$, we just need to update the original space $\mathcal{F}(\mathcal{D}_{org}, ms_{\%})$ iteratively based on Theorem 5.2 for each $t_i \in \mathcal{D}_{inc}$ $(1 \leq i \leq n)$.

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 3 (a) is an example.

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 5.2; (3) discover newly emerged frequent equivalence classes; and (4) remove existing frequent equivalence classes that are no longer frequent. 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 focus here 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.

5.2 Maintenance Data Structure: Generator-Enumeration 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

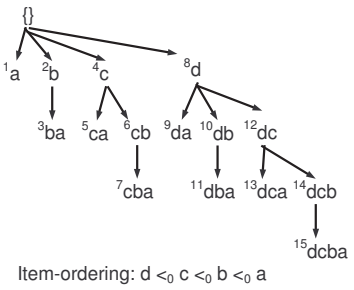

Figure 4: The Set-Enumeration Tree with item order: $d <_0 c <_0 b <_0 a$. The number on the left top corner of each node indicates the order at which the node is visited.

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.

5.2.1 Set-Enumeration Tree

Set-Enumeration Tree (*SE-tree*), as shown in Figure 4, 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 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 4 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 4. 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.

5.2.2 Generator-Enumeration Tree

The *Generator-Enumeration Tree* (*GE-tree*) is developed from the *SE-tree*. As shown in Figure 5 (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 (Wang *et al.*, 2000), closed patterns (Wang *et al.*, 2003) and maximal patterns (Bayardo, 1998). 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, in *GE-tree*, items are 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 5, each node in *GE-tree* represents a generator, and each frequent generator is linked to its corresponding equivalence

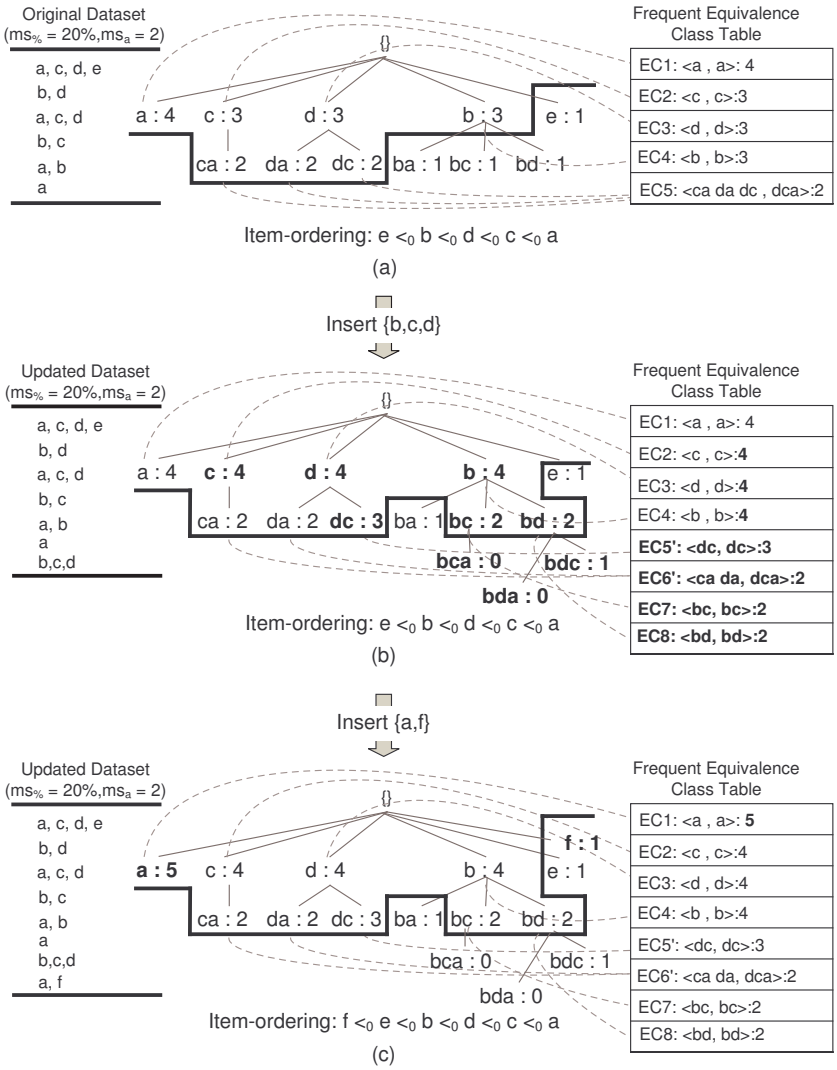

Figure 5: (a) The *GE-tree* for the original dataset. (b) The updated *GE-tree* when new transaction $\{b, c, d\}$ is inserted. (c) The updated *GE-tree* when new transaction $\{a, f\}$ is inserted.

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* in Figure 5, 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 $X \not\subseteq t_+$, according to Scenario 1 in Theorem 5.2, no update action is needed for X and its corresponding equivalence class. Furthermore, according to the "a priori" property of generators (Fact 4.6), all the children of X can be skipped for the traversal. For example, in Figure 5 (c), when transaction $\{a, f\}$ is inserted, only node $\{a\}$ needs to be updated and all the other nodes are skipped.

In addition, since *GE-tree* also stores the support information of frequent generators and negative border generators, the support of generators and their equivalence class can be updated without scanning of the original and incremental dataset. This greatly reduced the I/O overheads involved in PSM+. The support update process is graphically illustrated in Figure 5.

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 Scenario 3 and 4 in Theorem 5.2, we have the following corollary.

Corollary 5.3 *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 $\exists Q \in [P]_{\mathcal{D}_{org}}$ such that $Q \subseteq t_+$ but $Clo([P]_{\mathcal{D}_{org}}) \not\subseteq t_+$, where $Clo([P]_{\mathcal{D}_{org}})$ 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 $Clo(X) \not\subseteq t_+$. In Figure 5, 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 $Clo(EC5) = \{a, c, d\} \not\subseteq \{b, c, d\}$.

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.

5.2.3 Negative Generator Border

The "negative generator border" is defined based on the the idea of "negative border". The notion of negative border is first introduced in Mannila and Toivonen (1997). 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 5.4, refers to the set of infrequent generators that have frequent precedents in the *GE-tree*. In Figure 5, the generators immediately under the solid line are "negative border generators", and the collection of all these generators forms the "negative generator border".

Definition 5.4 (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 5, 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 5 (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 points 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 5 (b). We discover that, after new transactions are added, the negative generator border expands and moves away from the root of *GE-tree*.

Procedure 1 enumNewEC

Input: G, 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 G.support ≥ ms_a then
2:    //Newly emerged frequent generator and equivalence class.
3:    Let C be the corresponding closed pattern of G;
4:    if ∃EC ∈ F such that EC.close = C then
5:       G → EC.keys;
         {The corresponding equivalence class already exists.}
6:    else
7:       Create new equivalence class EC';
8:       EC'.close = C, G → EC'.keys;
9:       EC' → F;
10:   end if
      {Enumerate new generators from G}
11:   for all X, where X is the left hand side sibling of G in GE-tree do
12:      G' := G ∪ X;
13:      if G' is a generator then
14:         enumNewEC(G', F, ms_a, GE-tree);
15:      end if
16:   end for
17: else
18:   G → GE-tree.ngb; {New negative generator border.}
19: end if
20: return F and GE-tree;
```

The detailed enumeration process is presented in Procedure 1. In Procedure 1 and all subsequent pseudo-codes, the following notations are used: $X.support$ denotes the support of pattern/equivalence class X; $X.close$ refers to the closed pattern of equivalence class X; $X.keys$ refers to the generators of equivalence class X; *GE-tree.ngb* refers to the negative generator border of the *GE-tree* and $X \rightarrow Y$ denotes the insertion of X into Y.

Procedure 1 is called at a starting point node G in the current negative generator border. If G is frequent (Line 1), then it is a newly emerged frequent generator. If its equivalence class EC has already been created (Lines 3-4), we simply include G into EC's set of generators (Line 5). Otherwise, we create the new frequent equivalence class EC' corresponding to G (Lines 6-10). Finally, we recurse on the children of G (Lines 11-16). On the other hand, if G is not frequent, then we insert G into the negative generator border and halt the enumeration (Line 17-19).

Algorithm 2 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.ngb*; {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 5.2.}
13: **else**
14: splitEC($\mathcal{F}$, t, G); {split up EC.} {Corresponds to Scenario 3 & 4 of Theorem 5.2.}
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 5.2.}
18: **end if**
19: **end for**
20: **end for**
21: Include the frequent equivalence classes in $\mathcal{F}$ into $\mathcal{F}_{upd+}$;
22: **return** $\mathcal{F}_{upd+}$ and the updated *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.

5.3 Proposed Algorithm: PSM+

A novel incremental maintenance algorithm, *Pattern Space Maintainer+* (PSM+), is proposed based on the *GE-tree*. The pseudo-code of PSM+ is presented in Algorithm 2, Procedure 1 and Procedure 3.

Algorithm 2 maintains the frequent pattern space by considering only one incremental transaction at a time (Line 3). If the incremental transaction contains some new items (Line 4), Algorithm 2 starts off by inserting these new items, as singleton generators[4], into the negative generator border of *GE-tree*

[4]Singleton generators refer to generators that contain only one item.

Procedure 3 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: $G \rightarrow EC''.keys$;
7: **else**
8: Create new equivalence class EC';
9: $EC'.close = C_{new}$, $EC'.support = G.support$, $G \rightarrow EC'.keys$;
10: $EC' \rightarrow \mathcal{F}$;
11: **end if**
12: **return** $\mathcal{F}$;

(Line 5). Next, for each generator G in the *GE-tree* that is contained in the incremental transaction (Line 7), we first updates its support (Line 8). Then, we have two cases. In the first case, G is an existing frequent generator (Line 9). In this case, we carry on to update EC, the corresponding equivalence class of G (Line 10-15). If the closed pattern of EC is subset of the incremental transaction, the maintenance is simple. We just need to update the support of EC (Line 12). Otherwise, equivalence class EC needs to be split into two as described in Procedure 3. In the second case, G is a newly emerged frequent generator. In this case, the update of G is handled by Procedure 1 as described in the previous section. Finally, the updated frequent pattern space is formed with all the updated and newly generated frequent equivalence classes (Line 21).

Theorem 5.5 *PSM+ presented in Algorithm 2 correctly maintains the frequent pattern space, which is represented using equivalence classes, for incremental updates.*

Proof: *Refer to Appendix.* □

5.3.1 A Running Example

We demonstrate how PSM+ updates the frequent pattern space with the example shown in Figure 5. In Figure 5, 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 updated absolute support threshold $ms_a = \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil = 2$ (Line 2 of Algorithm 2). For each incremental transaction, PSM+ updates the affected equivalence classes through updating their corresponding generators. In Figure 5 (b) and (c), the affected generators and equivalence classes are highlighted in bold.

We further illustrate in detail how PSM+ addresses different maintenance scenarios with a few representative examples. First, we investigate the scenario, where only the support of the corresponding equivalence class needs to be updated. Suppose incremental transaction $\{b, c, d\}$ is inserted, and let us consider generator $\{c\}$ as an example. Since $\{c\} \subseteq \{b, c, d\}$ (Line 7 of Algorithm 2), $\{c\}$ is an affected generator. The support of $\{c\}$ is then updated by Line 8. Also Since $\{c\}$ is an existing frequent generator (Line 9), we carry on to update its corresponding equivalence class, $EC2$. As shown in Figure 5 (b), the closed pattern of $EC2$ is also $\{c\}$. Thus, we have $EC2.close \subseteq \{b, c, d\}$ (Line 11). Therefore, 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. 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 5 (a), and, more importantly, $EC5.closed = \{d, c, a\} \not\subseteq \{b, c, d\}$.

Therefore, Line 11 of Algorithm 2 is not satisfied. Thus, as desired, $EC5$ will be split into two as described in Procedure 3. As shown in Figure 5 (b), $EC5$ splits into $EC5'$ and $EC6'$. In Procedure 3, $EC6'$ is considered as the first split out of $EC5$, and it is updated by Line 2 of Procedure 3. 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 5 (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 2 is not satisfied. However, as highlighted in Figure 5 (b), generator $\{b, c\}$ becomes frequent after the update (Line 16 of Algorithm 2). Thus, the corresponding equivalence class $EC7$ is then included as frequent equivalence class by Line 1 to 11 of Procedure 1. Furthermore, $\{b, c\}$ also acts as a starting point for further enumeration of new generators as stated in Line 12 to 19 of Procedure 1.

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 (Line 4 of Algorithm 2). Therefore, as illustrated in Figure 5 (c), after the insertion of transaction $\{a, f\}$, generator $\{f\}$ is inserted into the *GE-tree* (Line 5 of Algorithm 2) as negative border generator. 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.

5.3.2 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. Re-

dataset	#PSM+	#FPgrowth*	#GC-growth
bms-pos ($ms_{\%} = 0.1\%$)	80	110K	110K
bms-webview1 ($ms_{\%} = 0.1\%$)	250	3K	3K
chess ($ms_{\%} = 40\%$)	350K	6M	1M
connect-4 ($ms_{\%} = 20\%$)	80K	1800M	1M
mushroom ($ms_{\%} = 0.5\%$)	10K	300M	165K
pumsb* ($ms_{\%} = 30\%$)	2K	400K	27K
retail ($ms_{\%} = 0.1\%$)	270	8K	8K
T10I4D100K ($ms_{\%} = 0.5\%$)	11	1K	1K
T40I10D100K ($ms_{\%} = 10\%$)	7K	70K	55K

Table 1: Comparison of the number of patterns enumerated by PSM+, FP-growth* and GC-growth. Notations: #PSM+, #FPgrowth* and #GC-growth denote the approximated number of patterns enumerated by the respectively algorithms.

call that the incremental maintenance of frequent patterns involves three major computational tasks: the support update task, class splitting task and new class discovery task. We have demonstrated that, with the help of *GE-tree*, the support update task and the class splitting task can be efficiently completed with little computational overhead. Therefore, the major contribution to the time complexity of PSM+ comes from the new class discovery task. For the new class discovery task, the time complexity is proportional to the number of patterns enumerated. As a result, the time complexity of PSM+ can be approximated as $O(N_{enum})$, where N_{enum} is the number of patterns enumerated. We have conducted some experiments to compare the number of patterns enumerated by PSM+ with the ones of FPgrowth* and GC-growth. FPgrowth* is one of the fastest frequent pattern discovery algorithms (Goethals and Zaki, 2003), and GC-growth is one of the fastest discovery algorithms for frequent equivalence classes (Li *et al.*, 2005). 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 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

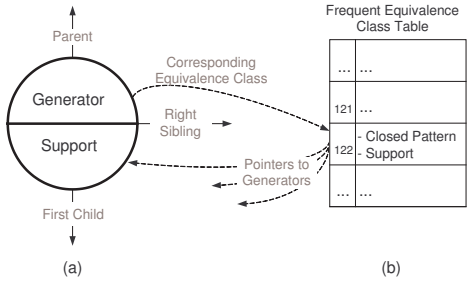

Figure 6: (a) Showcase of a *GE-tree* node. (b) The frequent equivalence class table, highlighting the corresponding equivalence class of the *GE-tree* node in (a).

complexity, PSM+ is much more effective than FPgrowth* and GC-growth.

5.3.3 Implementation Details

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 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 5 and 6 (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 (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 their closed patterns and generators.

We employ GC-growth[5] (Li *et al.*, 2005) 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 borders of equivalence classes, the generation of corresponding closed patterns is required. A prefix tree structure, named *mFP-tree*, is developed for this task.

mFP-tree is a modification of *FP-tree* (Han *et al.*, 2000), which is the representative prefix tree that concisely summarizes transactional datasets. The key features and construction of *FP-tree* can be referred to (Han *et al.*, 2000). We emphasize here the two major modifications in *mFP-tree*. (1) In *FP-tree*, only frequent items of each transaction are recorded, but, in *mFP-tree*, all items are recorded. E.g. in Figure 7 (b), although item e is not a frequent item, it is still recorded in the *mFP-tree*. This modification allows *mFP-tree* to be updated without re-scanning of the original dataset. (2) Items in *mFP-tree* are sorted based on their support values in the original dataset. More importantly, the ordering of items remains unchanged for all subsequent updates. This fixed ordering of items, as demonstrated in Figure 7 (c), allows new transactions to

[5]The implementation of GC-growth can be found in `http://www.comp.nus.edu.sg/~wongls/projects/pattern-spaces/gcgrowth-v1/`.

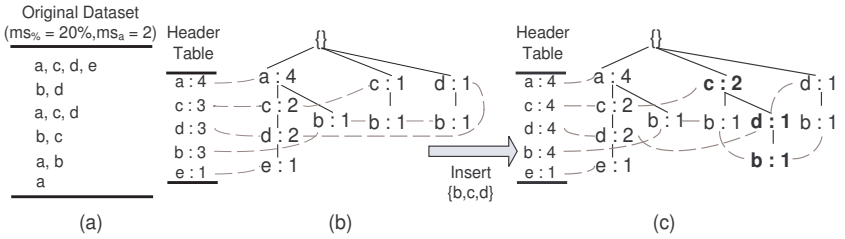

Figure 7: (a) A sample data set with $ms_{\%} = 20\%$ and $ms_a = 2$. (b) The *mFP-tree* for the dataset in (a). (c) The updated *mFP-tree* after the insertion of transaction $\{b, c, d\}$.

be inserted into *mFP-tree* easily without any re-sorting and swapping of nodes.

With *mFP-tree*, the generation of closed patterns for newly emerged frequent generators becomes straightforward. We use the *mFP-tree* in Figure 7 (c) as an example. Suppose we want to find 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 $\{b\}$.

The original *mFP-tree* is generated with GC-growth in the initialization phase. Since GC-growth also employs *mFP-tree* to enumerate frequent generators and closed patterns, no extra overhead is introduced. Moreover, the *mFP-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 *mFP-tree* without re-visiting the original dataset.

Note that: although the above implementation techniques are discussed in the context of PSM+, they are also employed in PSM- to facilitate the maintenance process.

6 Decremental Maintenance of Pattern Space

This section discusses the decremental maintenance of the frequent pattern space. 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 $ms_{\%}$, the task of decremental maintenance is to obtain the updated pattern space by maintaining the original pattern space.

To develop effective decremental maintenance algorithm, we start off with a study on the evolution of the frequent pattern space under decremental updates using the concept of equivalence class. Through the space evolution study, we summarize the major computational tasks in the decremental maintenance. We then demonstrate how these computational tasks can also be completed efficiently using *GE-tree*. Finally, a novel decremental maintenance algorithm, named *Pattern Space Maintainer-* (PSM-), is proposed.

6.1 Evolution of Pattern Space

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 resulting dataset 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 resulting dataset with $\mathcal{D}_{inc} = \mathcal{D}_{dec}$, we get back the original dataset. Therefore, the decremental maintenance is actually the reverse process of incremental maintenance.

After an incremental update, new frequent equivalence classes may emerge; in contrast, existing frequent equivalence classes may become infrequent after a decremental update. Moreover, for those existing frequent equivalence classes that are still frequent after the decremental update, they may evolve in three different ways. 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 merge with other classes. We know from Proposition 5.1 that an equivalence class may shrink in size and increase in support after an incremental update. It follows by duality that an equivalence class may increase in size (by merging) and decrease in support after a decremental update.

Corollary 6.1 *Let P be a pattern in $\mathcal{D}_{upd-}$. Then $[P]_{\mathcal{D}_{upd-}} \supseteq [P]_{\mathcal{D}_{org}}$, and $sup(P, \mathcal{D}_{upd-}) \leq sup(P, \mathcal{D}_{org})$.*

To have a deeper understanding on how the frequent pattern space evolves under the decremental update, we investigate the exact conditions for each evolution scenario to occur. We denote the closed pattern of an equivalence class $[p]_{\mathcal{D}}$ as $Clo([p]_{\mathcal{D}})$ and the generators or key patterns of $[p]_{\mathcal{D}}$ as $Keys([p]_{\mathcal{D}})$.

Theorem 6.2 *Let $\mathcal{D}_{org}$ be the original dataset, $\mathcal{D}_{dec}$ be the decremental dataset, $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \mathcal{D}_{dec}$ and $ms_{\%}$ be the support threshold. For simplicity, we assume $\mathcal{D}_{dec}$ consists of only one transaction t_-. 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. *$P \notin \mathcal{D}_{dec}$ 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. *$P \notin \mathcal{D}_{dec}$ 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. Let all such Q's 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 $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup \bigcup_i [Q_i]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$, $Clo([P]_{\mathcal{D}_{upd-}}) = Clo([P]_{\mathcal{D}_{org}})$ and $Keys([P]_{\mathcal{D}_{upd-}}) = min\{K | K \in Keys([P]_{\mathcal{D}_{org}}) \vee K \in$*

$Keys([Q_i]_{\mathcal{D}_{org}}), 1 \leq i \leq n\}$. *Furthermore,* $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$, *and* $[Q_i]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{upd-}}$ *for* $1 \leq i \leq n$.

3. $P \in \mathcal{D}_{dec}$ *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. $P \in \mathcal{D}_{dec}$, $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ *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,* $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org}) - sup(P, \mathcal{D}_{dec})$ *and* $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$.

5. $P \in \mathcal{D}_{dec}$, $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ *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: *Refer to Appendix.* □

Theorem 6.2 summarizes how the frequent pattern space evolves 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 5.2, Theorem 6.2 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.

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 Scenario 2 and 5 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 excluded from our discussion, for its solution is straightforward. 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 Maintenance of Pattern Space

We investigate here how the major computational tasks in decremental maintenance of the frequent pattern space can be efficiently accomplished.

Due to the duality between the incremental and decremental maintenance, most of the computational tasks in decremental maintenance can be effectively handled with the *GE-tree*. In particular, the **support update** task in decremental maintenance is actually 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 5.2.2. Except that, in decremental maintenance, the support is decremented.

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 1 in Section 5.2.3. Same as in incremental maintenance, the negative generator border is updated after the removal of each old transactions. However,

different from incremental updates, when old transactions are removed, the negative generator border shrinks and move towards the root of *GE-tree*.

On the other hand, the **class merging** task can not be handled in the same way as the class splitting task in incremental maintenance. However, extended from the Scenario 2 in Theorem 6.2, we have the following corollary.

Corollary 6.3 *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)* $Clo([P]_{\mathcal{D}_{org}}) \supset Clo([Q]_{\mathcal{D}_{org}})$. *Here* $Clo(X)$ *denotes the closed pattern of equivalence class* X.

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 (ii) 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,* $Clo([P]_{\mathcal{D}_{org}}) \supset Clo([Q]_{\mathcal{D}_{org}})$.

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

Corollary 6.3 provides us a means to determine which two equivalence classes need to be merged after an decremental update. Based on Corollary 6.3, 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 su-

Algorithm 4 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: $\mathcal{F}_{upd-} := \mathcal{F}$
30: **return** $\mathcal{F}_{upd-}$ and the updated *GE-tree*;

perset and subset to each other. Details of this merging process is presented in Algorithm 4, which will be discussed in the next section.

6.3 Proposed Algorithm: PSM-

A novel algorithm, *Pattern Space Maintainer-* (PSM-), is proposed for the decremental maintenance of the frequent pattern space. The pseudo-code of PSM- is presented in Algorithm 4 and Procedure 1. In Algorithm 4 and Procedure 1, we use notations: $X.support$ to denote the support of pattern/equivalence class X; $X.close$ to denote the closed pattern of equivalence class X; $X.keys$ to denote the set of generators of equivalence class X and

$X \rightarrow Y$ to denote the insertion of X into Y. Algorithm 4 begins with the support-update phase. For each transaction in the decremental dataset (Line 3), and for each existing generator that is contained in the transaction (Line 4), we update the support of the generator and its equivalence class (Lines 5-9). If the generator becomes infrequent, we move the negative generator border towards it (Lines 10-13). After this update phase is completed, we inspect the negative generator border to enumerate newly emerged frequent generators (Line 16-18). Finally, we inspect all new and existing equivalence classes to merge those frequent equivalence classes that should be merged (Lines 19-24) and to remove those that have become infrequent (Lines 25-27).

Theorem 6.4 *PSM- presented in Algorithm 4 correctly maintains the frequent pattern space, which is represented using equivalence classes, for decremental updates.*

Proof: *Refer to Appendix.* □

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.

7 Pattern Space Maintainer (PSM)

We have proposed a novel algorithm, PSM+, to address the incremental maintenance of the frequent pattern space, and we have also proposed a novel algorithm, PSM-, for the decremental maintenance. Although these two maintenance algorithms are discussed separately, PSM+ and PSM- share many similarities and are both developed based on the same data structure — the *GE-tree*. Thus the integration of PSM+ and PSM- involves negligible overheads. We name the integrated version of PSM+ and PSM- the *Pattern Space Maintainer*, in short PSM.

PSM is not only a useful tool for incremental and decremental maintenance, it can also be employed to maintain the space of frequent patterns for support threshold adjustment. Support threshold adjustment is a common interactive mining operation, which is used to obtain the appropriate set of frequent patterns. When the support threshold is adjusted up, existing frequent patterns and equivalence classes may become infrequent. The maintenance for this scenario is very straightforward, and thus we will not discuss it here. On the other hand, when the support threshold is adjusted down, new (unknown) frequent patterns and equivalence classes may emerge. The maintenance for this scenario is much more challenging, for we have little information on the newly emerged patterns. In this case, PSM can be used to effectively enumerate the newly emerged equivalence classes based on the concepts of *GE-tree* and negative generator border. The detailed enumeration method is described in Procedure 1 in Section 5.2.3.

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 2: Characteristics of testing datasets. 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.

8 Experimental Studies

The computational effectiveness of the proposed algorithms is tested on the benchmark datasets from the *FIMI* Repository (`http://fimi.cs.helsinki.fi`). The statistical information of the benchmark datasets is summarized in Table 2. The benchmark datasets include 10 real datasets and 2 synthetic datasets. Experiments were run on a PC with 2.4GHz CPU and 3.2G of memory. The proposed algorithms are implemented in C_{++}.

The performance of the proposed algorithms are compared with the state-of-the-art approaches, including: FPgrowth* (Grahne and Zhu, 2005), one of the fastest frequent pattern discovery algorithms; GC-growth (Li *et al.*, 2005), the fastest discovery algorithm for frequent equivalence classes; CanTree (Leung *et al.*, 2007), a prefix-tree based maintenance algorithm; moment (Chi *et al.*, 2006), a currently proposed algorithm that maintains frequent closed patterns; and ZIGZAG (Veloso *et al.*, 2002), a frequent maximal pattern maintenance algorithm.

Incremental Maintenance

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 $\Delta^+ = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$. In addition, we observe that the performance of algorithms varies slightly for different combinations of incremental and original datasets. To have a stable performance measurement, for each update interval, 5 random sets of transactions were first removed from the testing datasets: the removed set of transactions was treated as the incremental dataset, $\mathcal{D}_{inc}$, and the remaining set of transactions was treated as the original dataset, $\mathcal{D}_{org}$. The average performance over the 5 random combinations was then recorded. This averaging strategy is applied in all experimental studies.

Figure 8 compares the performance of PSM+ with the discovery algorithms, GC-growth and FPgrowth*. 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. The detailed computational "speed up" achieved by PSM+ is summarized in Table 3. 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.

PSM+ is also compared with the state-of-the-art maintenance algorithms, which includes CanTree, moment and ZIGZAG. Some representative results are

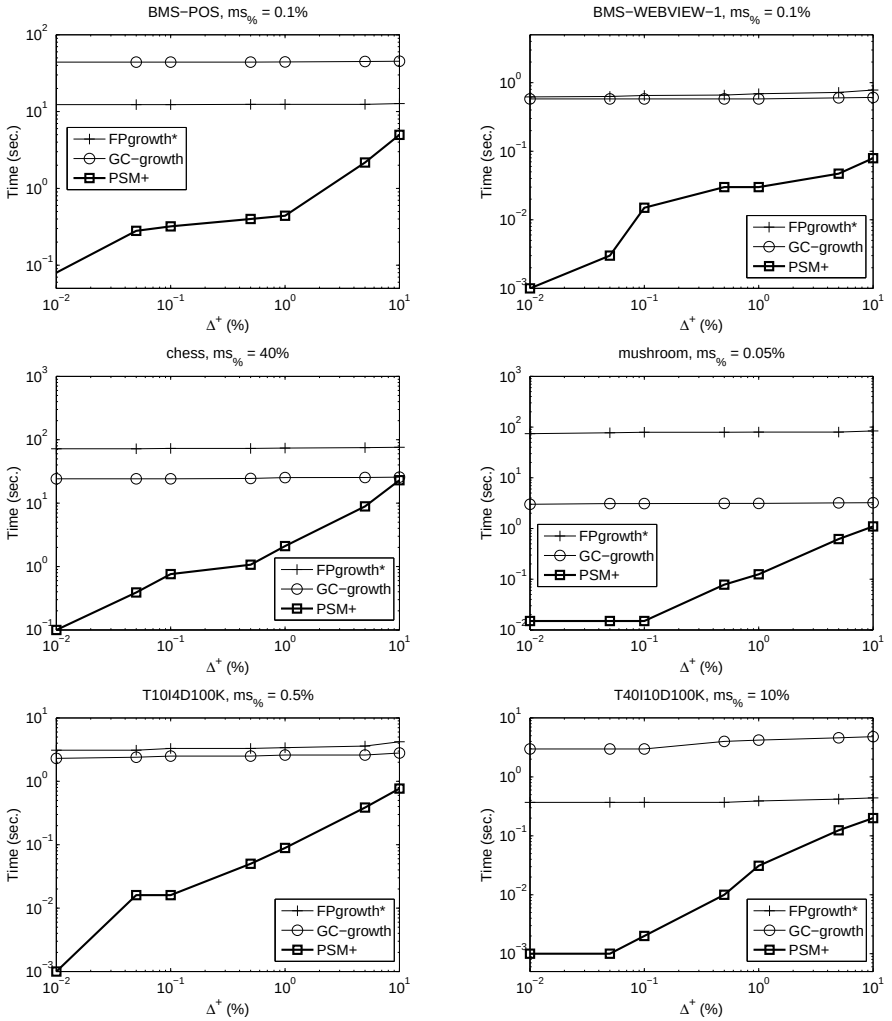

Figure 8: Performance comparison of PSM+ and the pattern discovery algorithms: FPgrowth* and GC-growth. Notations: $\Delta^+ = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

graphically presented in Figure 9. According to the empirical results, PSM+ is the most effective algorithm among all. Take dataset *mushroom* as an example. PSM+ is more than an order of magnitude faster than CanTree, and, compared with moment and ZIZAG, it is faster by almost 2 orders of magnitude. The average "speed up" of PSM+ against the maintenance algorithms is also summarized in Table 3. PSM+, on average, outperforms moment and ZIGZAG by

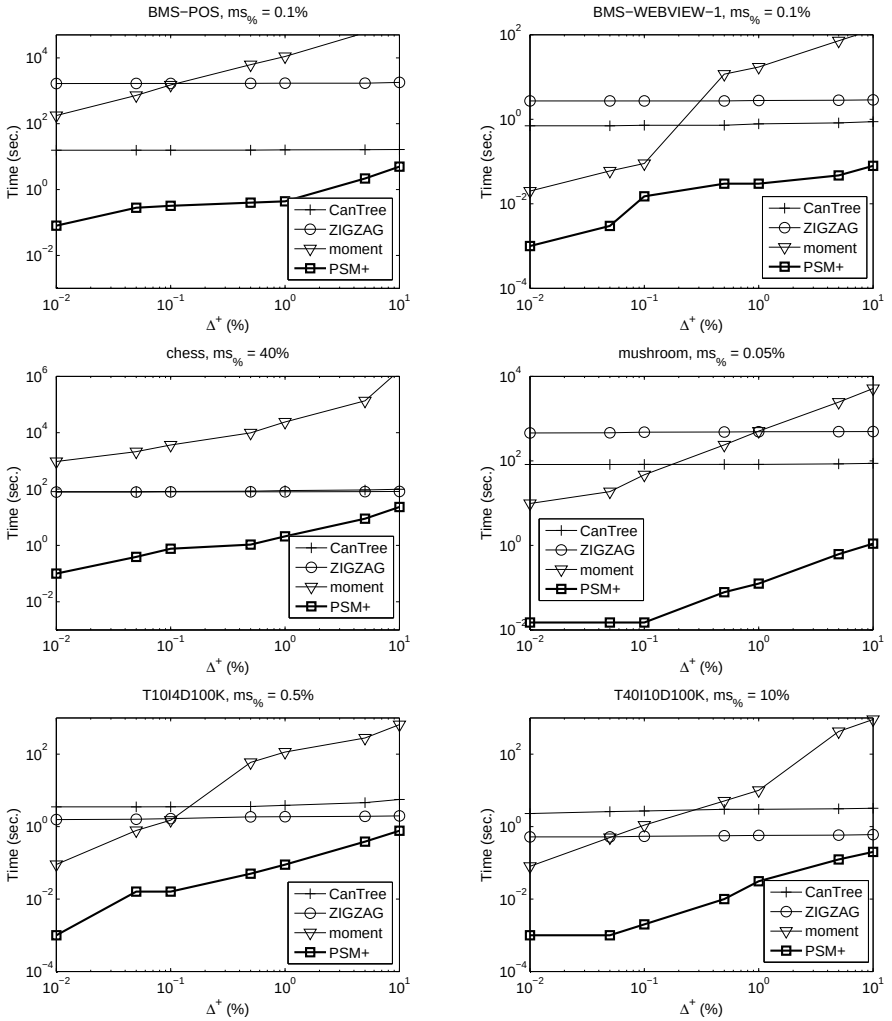

Figure 9: Performance comparison of PSM+ and the pattern maintenance algorithms, CanTree, ZIGZAG and moment. Notations: $\Delta^+ = |\mathcal{D}_{inc}|/|\mathcal{D}_{org}|$.

more than 3 orders of magnitude and outperforms CanTree by over 700 times.

Decremental Maintenance

With the similar reason of incremental maintenance, the performance of PSM- is evaluated for $\Delta^- \leq 10\%$, where $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$. The performance of PSM- is also compared with both pattern discovery and pattern maintenance algorithms, as shown in Figure 10.

Dataset	Discovery Algorithms		Maintenance Algorithms		
	FPgrowth*	GC-growth	CanTree	ZIGZAG	moment
accidents (50%)	12.5	76	15	270	22
accidents (40%)	1.6	9.5	2	56.5	6.2
BMS-POS (0.1%)	43	155	55	5,880	14,400
BMS-POS (0.5%)	**126**	**390**	**130**	**13,500**	**23,000**
BMS-WEBVIEW-1 (0.1%)	136	125	152	588	741
BMS-WEBVIEW-1 (0.05%)	963	370	1,015	672	75
BMS-WEBVIEW-2 (0.05%)	35	96	40	1,900	715
BMS-WEBVIEW-2 (0.01%)	**1,300**	**316**	**1,420**	**13,000**	**615**
chess (50%)	590	96	620	1,395	13,000
chess (40%)	169	18	180	172	18,100
connect-4 (50%)	2,280	8.2	2340	1,400	826
connect-4 (45%)	2,740	5.6	2,810	1,800	824
mushroom (0.1%)	**3,085**	**380**	**3,121**	**47,800**	**3,216**
mushroom (0.05%)	2,457	81	2,630	15,000	2,960
pumsb (70%)	1.6	1.5	1.8	6.9	1,662
pumsb (60%)	3.5	23.5	3.8	16.5	640
pumsb_star (50%)	101	420	123	25.7	7,540
pumsb_star (40%)	3.6	20	7.2	16	2,970
retail (0.1%)	640	247	735	27,100	18,210
retail (0.05%)	**985**	**98.5**	**1,050**	**38,500**	**28,340**
T10I4D100K (0.5%)	150	374	200	261	609
T10I4D100K (0.05%)	41	64	45.5	120	81
T40I10D100K (10%)	140	1,145	955	102	1,415
T40I10D100K (5%)	138	**1,777**	269	36	1,118
Average	672	262	746	7,067	5,878

Table 3: Average speed up of PSM+ over benchmark datasets. The percentage in brackets after the dataset name indicates the minimum support threshold.

As illustrated in Figure 10 (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

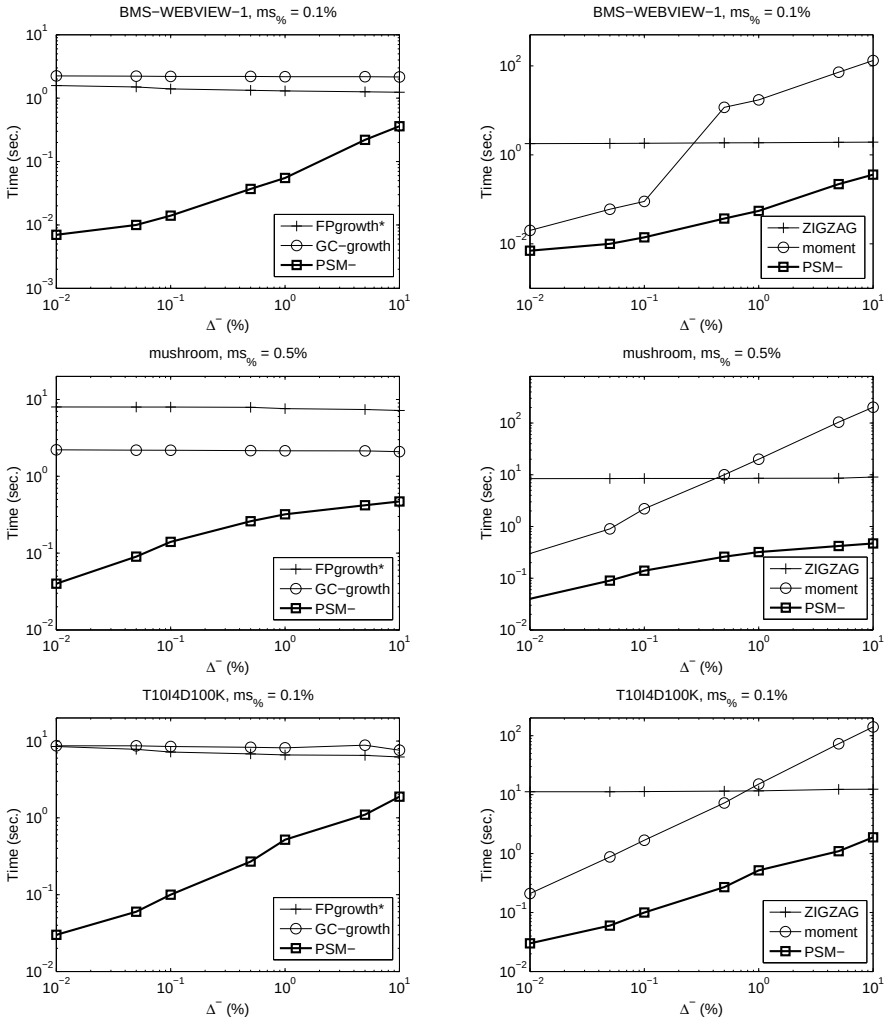

Figure 10: (a) Performance comparison of PSM- and the pattern discovery algorithms: FPgrowth* and GC-growth. (b) Performance comparison of PSM- and the pattern maintenance algorithms: ZIGZAG, moment and TRUM. Notations: $\Delta^- = |\mathcal{D}_{dec}|/|\mathcal{D}_{org}|$.

up to 10%, PSM- is still 5 times more efficient. Table 4 summarizes the average "speed up" achieved by PSM-. 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*

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

Table 4: Average speed up of PSM- over benchmark datasets. The percentage in the brackets after the dataset name indicates the minimum support threshold.

and $T10I4D100K$, where PSM- runs over 300 times faster. On average, PSM- outperforms both discovery algorithms by around 2 orders of magnitude.

Figure 10 (b) graphically compares PSM- with other maintenance algo-

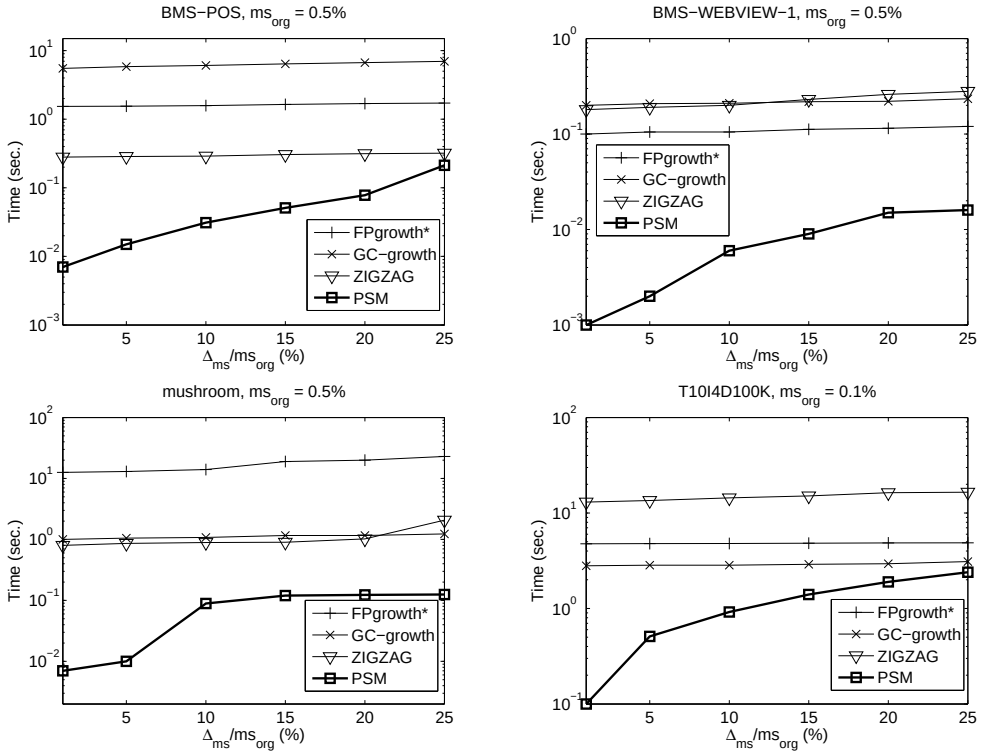

Figure 11: Performance comparison of PSM and the discovery algorithms, FPgrowth* and GC-growth, and the maintenance algorithm, ZIGZAG. Notations: Δ_{ms} denotes the difference between the original support threshold and the updated support threshold.

rithms. Compared with moment and ZIGZAG, PSM-, in most cases, is at least 10 times faster. According to Table 4, PSM-, on average, outperforms ZIGZAG by almost 800 times and outperforms moment by almost 4 orders of magnitude.

Support Adjustment Maintenance

We have also evaluated the performance of PSM for support threshold adjustment. The effectiveness of PSM is tested with various degrees of threshold adjustment. The experimental results are presented in Figure 11. As can been seen from Figure 11, PSM outperforms both the pattern discovery and pattern maintenance algorithms considerably.

Discussions

We observe that, over three different types of updates, our proposed algorithms outperform the discovery algorithms by multiple orders of magnitude. This is mainly due to three advantages of our algorithms. First, we structurally decomposed the vast frequent pattern space into equivalence classes. The structure decomposition greatly simplifies the complexity of the maintenance problem and allows us to address the problem in a divide-and-conquer manner. Second, with *GE-tree*, our algorithms effectively maintain the frequent pattern space by updating only the equivalence classes that are affected by the updates. Third, as demonstrated in Table 1, while generating new frequent equivalence classes, our algorithms enumerate much less candidates compared with the discovery algorithms.

We also observe that the advantage of the proposed algorithms diminishes as the size (or degree) of update increases. This is because large update size or large variation in support threshold logically leads to more dramatic changes to the frequent pattern space and makes the pattern space computationally more expensive to be maintained. 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.

9 Conclusion

This paper has studied the incremental and decremental maintenance of the frequent pattern space. To develop efficient maintenance algorithms, we started off by analyzing how the space of frequent patterns evolves under incremental and decremental updates. Since the frequent pattern space is too huge to be analyzed directly, we structurally decomposed the pattern space into convex

equivalence classes. The structure decomposition allows us to formally describe the evolution of frequent pattern space and also greatly simplifies the maintenance problem. Based on this space evolution analysis, we have summarized the major computation tasks involved in frequent pattern maintenance. To effectively perform the maintenance computational tasks, a new data structure, *Generator-Enumeration Tree* (*GE-tree*), is developed. Based on *GE-tree*, we proposed two novel algorithms, *Pattern Space Maintainer+* (PSM+) and *Pattern Space Maintainer-* (PSM-), for the incremental and decremental maintenance of frequent patterns. We further demonstrated that PSM+ and PSM- can be easily integrated and extended to update the frequent pattern space for support threshold adjustment. We have evaluated the effectiveness of our proposed algorithms with extensive experimental studies. Experimental results show that the proposed algorithms on average outperform the state-of-the-art approaches by at least an order of magnitude.

This paper studied the evolution of the frequent pattern space. In the future, we plan to explore the evolution and maintenance of other types of pattern spaces, e.g. the space of emerging patterns, odds ratio patterns, etc.

Acknowledgements

Thanks to Yun Chi from NEC Laboratories America, Inc. for the source code of moment. Thanks to Mohammed Javeed Zaki from Rensselaer Polytechnic Institute, USA, for the source code of ZIGZAG. This work was supported in part by an A*STAR AGS scholarship and A*STAR SERC PSF grant 072 101 0016.

References

Agrawal,R. and Imielinski,T. (1993) Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* pp. 207–216.

Bayardo,R.J. (1998) Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* pp. 85–93.

Cheung,D.W.L., Han,J., Ng,V.T.Y. and Wong,C.Y. (1996) Maintenance of discovered association rules in large databases: an incremental updating technique. In *Proceedings of the Twelfth International Conference On Data Engineering* pp. 106–114.

Cheung,D.W.L., Lee,S.D. and Kao,B. (1997) A general incremental technique for maintaining discovered association rules. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications* pp. 185–194.

Chi,Y., Wang,H., Yu,P.S. and Muntz,R.R. (2006) Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. *Knowledge and Information Systems,* **10** (3), 265–294.

Feng,M., Dong,G., Li,J., Tan,Y.P. and Wong,L. (2009) Evolution of frequent pattern space. *International Journal of Foundations of Computer Science,* **submitted** (n.a.), n.a.

Goethals,B. and Zaki,M.J. (2003) Advances in frequent itemset mining implementations: introduction to fimi03. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations.*

Gouda,K. and Zaki,M.J. (2001) Efficiently mining maximal frequent itemsets. In *Proceedings of the 2001 IEEE International Conference on Data Mining* pp. 163–170.

Grahne,G. and Zhu,J. (2005) Fast algorithms for frequent itemset mining using fp-trees. *IEEE Transactions on Knowledge and Data Engineering,* **17** (10), 1347–1362.

Han,J., Pei,J. and Yin,Y. (2000) Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* pp. 1–12.

Lee,C.H., Lin,C.R. and Chen,M.S. (2005) Sliding window filtering: an efficient method for incremental mining on a time-variant database. *Information Systems,* **30** (3), 227–244.

Leung,C.K.S., Khan,Q.I., Li,Z. and Hoque,T. (2007) Cantree: a canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems,* **11** (3), 287–311.

Li,C., Cong,G., Tung,A.K.H. and Wang,S. (2004) Incremental maintenance of quotient cube for median. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* pp. 226–235.

Li,H., Li,J., Wong,L., Feng,M. and Tan,Y.P. (2005) Relative risk and odds ratio: a data mining perspective. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* pp. 368–377.

Mannila,H. and Toivonen,H. (1997) Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery,* **1** (3), 241–258.

Pasquier,N., Bastide,Y., Taouil,R. and Lakhal,L. (1999) Discovering frequent closed itemsets for association rules. In *Proceedings of Seventh International Conference of Data Theories* pp. 398–416.

Veloso,A., Jr.,W.M., de Carvalho,M., Pôssas,B., Parthasarathy,S. and Zaki,M.J. (2002) Mining frequent itemsets in evolving databases. In *Proceedings of the Second SIAM International Conference on Data Mining.*

Wang,J., Han,J. and Pei,J. (2003) Closet+: searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* pp. 236–245.

Wang,K., He,Y. and Han,J. (2000) Mining frequent itemsets using support constraints. In *Proceedings of 26th International Conference on Very Large Data Bases* pp. 43–52.

Appendix: Proofs of Theorems [6]

Theorem 5.2 *Let $\mathcal{D}_{org}$ be the original dataset, $\mathcal{D}_{inc}$ be the incremental dataset, $\mathcal{D}_{upd+} = \mathcal{D}_{org} \cup \mathcal{D}_{inc}$ and $ms_{\%}$ be the support threshold. Suppose $\mathcal{D}_{inc}$ consists of only one transaction t_+. 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 $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + 1$.*

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. $[P]_{\mathcal{D}_{upd+}} = \{Q | Q \in [P]_{\mathcal{D}_{org}} \wedge Q \subseteq t_+\}$, $Clo([P]_{\mathcal{D}_{upd+}}) = Clo([P]_{\mathcal{D}_{org}}) \cap t_+$ and $Keys([P]_{\mathcal{D}_{upd+}}) = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$.*

4. *$P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$, $P \not\subseteq t_+$ 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, $[P]_{\mathcal{D}_{upd+}} = \{Q | Q \in [P]_{\mathcal{D}_{org}} \wedge Q \not\subseteq t_+\}$, $Clo([P]_{\mathcal{D}_{upd+}}) = Clo([P]_{\mathcal{D}_{org}})$ and $Keys([P]_{\mathcal{D}_{upd+}}) = min\{\{K | K \in$*

[6] The appendices are attached at the back of the paper for revision purpose. If page limit is a concern, the appendices will be removed from the final manuscript and placed on our homepage.

$Keys([P]_{\mathcal{D}_{org}}) \wedge K \not\subseteq t_+\} \cup \{K' \cup \{x_i\}, i = 1, 2, \cdots | K' \in Keys([P]_{\mathcal{D}_{org}}) \wedge K' \subseteq t_+, x_i \in Clo([P]_{\mathcal{D}_{org}}) \wedge x_i \notin t_+\}\}$.

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 $sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + sup(P, \mathcal{D}_{upd+}) = sup(P, \mathcal{D}_{org}) + 1$.*

Proof: *Scenario 1 and 5 are obvious.*

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 $sup(P, \mathcal{D}_{upd+}) = |f(P, \mathcal{D}_{upd+})| = sup(P, \mathcal{D}_{org}) + 1$.

To prove Scenario 3, suppose (i) $P \in \mathcal{F}(ms_{\%})$, (ii) $P \subset t_+$, and (iii)$Q \not\subseteq t_+$ for some $Q \in [P]_{\mathcal{D}_{org}}$. Point (ii) implies that $f(P, \mathcal{D}_{upd+}) = f(P, \mathcal{D}_{org}) \cup \{t_+\}$. Also for patterns Q that satisfy point (iii), $f(Q, \mathcal{D}_{upd+}) = f(Q, \mathcal{D}_{org}) \neq f(P, \mathcal{D}_{upd+})$. This means $Q \notin [P]_{\mathcal{D}_{upd+}}$. According to Definition 4.3, $[P]_{\mathcal{D}_{upd+}} = \{P' | f(P, \mathcal{D}_{upd+}) = f(P', \mathcal{D}_{upd+})\} = \{P' | P' \in [P]_{\mathcal{D}_{org}} \wedge P' \subseteq t_+\}$, and $[Q]_{\mathcal{D}_{upd+}} = \{Q' | Q' \in [P]_{\mathcal{D}_{org}} \wedge Q' \not\subseteq t_+\}$. Since $[P]_{\mathcal{D}_{org}} = [P]_{\mathcal{D}_{upd+}} \cup [Q]_{\mathcal{D}_{upd+}}$ and $[P]_{\mathcal{D}_{upd+}} \cap [Q]_{\mathcal{D}_{upd+}} = \emptyset$, we say that, in this case, the equivalence class $[P]_{\mathcal{D}_{org}}$ splits into two.

Next, we prove $Clo([P]_{\mathcal{D}_{upd+}}) = Clo([P]_{\mathcal{D}_{org}}) \cap t_+$. Let $C = Clo([P]_{\mathcal{D}_{org}}) \cap t_+$. It is obvious that (1) $C \subseteq Clo([P]_{\mathcal{D}_{org}})$, (2) $C \subseteq t_+$ and (3) $C \supseteq P$ (for $P \subseteq t_+$). 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

$C \in [P]_{\mathcal{D}_{upd+}}$. *We then assume that there exists* C' *such that* $C' \supset C$ *and* $C' \in [P]_{\mathcal{D}_{upd+}}$. $C' \in [P]_{\mathcal{D}_{upd+}}$ *implies that* $C' \in [P]_{\mathcal{D}_{org}}$ *and* $C' \subseteq t_+$. $C' \in [P]_{\mathcal{D}_{org}}$ *further implies that* $C' \subseteq Clo([P]_{\mathcal{D}_{org}})$. *Then we have* $C' \subseteq Clo([P]_{\mathcal{D}_{org}})$ *and* $C' \subseteq t_+$, *and thus* $C' \subseteq C$ *(for* $C = Clo([P]_{\mathcal{D}_{org}}) \cap t_+$*). This contradicts with the initial assumption. Therefore,* $C \in [P]_{\mathcal{D}_{upd+}}$ *and there does not exist* C' *such that* $C' \supset C$ *and* $C' \in [P]_{\mathcal{D}_{upd+}}$. *According to Definition 4.4,* C *is the closed pattern of* $[P]_{\mathcal{D}_{upd+}}$.

Then we prove $Keys([P]_{\mathcal{D}_{upd+}}) = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$. *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* $X \in [P]_{\mathcal{D}_{upd+}}$. $X \in \mathcal{K}$ *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 4.4). Moreover,* $[P]_{\mathcal{D}_{upd+}} \subset [P]_{\mathcal{D}_{org}}$. *Therefore,* X *must also be one of the most "general" patterns in* $[P]_{\mathcal{D}_{upd+}}$. *This means that* $X \in Keys([P]_{\mathcal{D}_{upd+}})$ *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* $Y \in Keys([P]_{\mathcal{D}_{upd+}})$ *but* $Y \notin \mathcal{K}$. $Y \in Keys([P]_{\mathcal{D}_{upd+}})$ *means* $Y \in [P]_{\mathcal{D}_{upd+}}$. *According to the definition of* $[P]_{\mathcal{D}_{upd+}}$, *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 4.4). Since* $K' \subset Y$ *and* $Y \subseteq t_+$, $K' \subset t_+$, *which implies* $K' \in [P]_{\mathcal{D}_{upd+}}$. *Thus, according to Definition 4.4,* $Y \notin Keys([P]_{\mathcal{D}_{upd+}})$. *This contradicts with the initial assumption. Thus there does not exists pattern* Y *such that* $Y \in Keys([P]_{\mathcal{D}_{upd+}})$ *but* $Y \notin \mathcal{K}$. *Therefore, we have (B)* $\mathcal{K} \supseteq Keys([P]_{\mathcal{D}_{upd+}})$. *Combining results (A) and (B), we have* $Keys([P]_{\mathcal{D}_{upd+}}) = \mathcal{K} = \{K | K \in Keys([P]_{\mathcal{D}_{org}}) \wedge K \subseteq t_+\}$.

Scenario 4 is complementary to Scenario 3. The proof for the splitting of equivalence class in Scenario 4 follows exactly the same as in Scenario 3. The definitions of the closed pattern and generators for the equivalence class $[P]_{\mathcal{D}_{upd+}}$

follows from Definition 4.4.

Finally, we prove that Theorem 5.2 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* $P \not\subseteq t_+$ *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 Theorem 5.2 is sound and complete.*

□

Theorem 5.5 *PSM+ presented in Algorithm 2 correctly maintains the frequent pattern space, which is represented using equivalence classes, for incremental updates.*

Proof: *According to Theorem 5.2, 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)* $Q \not\subseteq t_+$ *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 3) as desired. In particular,* $[P]_{\mathcal{D}_{upd+}}$ *described in Scenario 3 corresponds to the "second split out" in Procedure 3, and it is updated in Line 3 to 11 of Procedure 3.*

For Scenario 4, suppose (i) $P \in \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$*, (ii)* $P \not\subseteq t_+$ *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 3) as desired. Being complement to Scenario 3,* $[P]_{\mathcal{D}_{upd+}}$ *described in Scenario 4 corresponds to the "first split out" in Procedure 3, and it is updated in Line 2 of Procedure 3.*

For Scenario 5, suppose (i) $P \notin \mathcal{F}(ms_{\%}, \mathcal{D}_{org})$*, (ii)* $P \subseteq t_+$ *and (iii)* $Sup(P, \mathcal{D}_{upd+}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd+}| \rceil$*. 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 1. In Line 3 to 11 of Procedure 1,* $[P]_{\mathcal{D}_{upd+}}$ *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 the GE-tree yet.*

Therefore, the new generators will be included into the negative generator border of GE-tree by Line 5. Then, the generators and the corresponding equivalence class are 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 29 is put in to remove equivalence classes that are no longer frequent. With that, the theorem is proven. □

Theorem 6.2 *Let $\mathcal{D}_{org}$ be the original dataset, $\mathcal{D}_{dec}$ be the decremental dataset, $\mathcal{D}_{upd-} = \mathcal{D}_{org} - \mathcal{D}_{dec}$ and $ms_{\%}$ be the support threshold. For simplicity, we assume $\mathcal{D}_{dec}$ consists of only one transaction t_-. 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. *$P \notin \mathcal{D}_{dec}$ 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. *$P \notin \mathcal{D}_{dec}$ 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. Let all such Q's 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 $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup \bigcup_i [Q_i]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$, $Clo([P]_{\mathcal{D}_{upd-}}) = Clo([P]_{\mathcal{D}_{org}})$ and $Keys([P]_{\mathcal{D}_{upd-}}) = min\{K | K \in Keys([P]_{\mathcal{D}_{org}}) \vee K \in Keys([Q_i]_{\mathcal{D}_{org}}), 1 \leq i \leq n\}$. Furthermore, $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$, and $[Q_i]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{upd-}}$ for $1 \leq i \leq n$.*

3. *$P \in \mathcal{D}_{dec}$ 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. $P \in \mathcal{D}_{dec}$, $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ *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,* $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}}$, $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org}) - sup(P, \mathcal{D}_{dec})$ *and* $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$.

5. $P \in \mathcal{D}_{dec}$, $sup(P, \mathcal{D}_{upd-}) \geq \lceil ms_{\%} \times |\mathcal{D}_{upd-}| \rceil$ *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: *Scenario 1 and 3 are obvious.*

We first prove Scenario 4. Suppose (i) $P \in \mathcal{D}_{dec}$, *(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 6.1, 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* $sup(P, \mathcal{D}_{upd-}) = |f(P, \mathcal{D}_{upd-})| = |f(P, \mathcal{D}_{org}) - f(P, \mathcal{D}_{dec})| = sup(P, \mathcal{D}_{org}) - sup(P, \mathcal{D}_{dec})$.

To prove Scenario 2, suppose (i) $P \notin \mathcal{D}_{dec}$ *(ii)* $f(Q, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{upd-})$ *for some* $Q \notin [P]_{\mathcal{D}_{org}}$. *Point (ii) implies that some new patterns* $Q \notin [P]_{\mathcal{D}_{org}}$ *will be included into* $[P]_{\mathcal{D}_{upd-}}$. *Moreover, for such Qs, according to Corollary 6.1,* $Q' \in [Q]_{\mathcal{D}_{upd-}}$ *for every pattern* $Q' \in [Q]_{\mathcal{D}_{org}}$. *Thus it is also true that* $Q' \in [P]_{\mathcal{D}_{upd-}}$ *for every* $Q' \in [Q]_{\mathcal{D}_{org}}$. *Therefore, we say that* $[Q]_{\mathcal{D}_{org}}$ *merge with* $[P]_{\mathcal{D}_{org}}$ *and* $[Q]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{upd-}}$. *Let all such Q's 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 we have $[P]_{\mathcal{D}_{upd-}} = [P]_{\mathcal{D}_{org}} \cup \bigcup_i [Q_i]_{\mathcal{D}_{org}}$.

Point (i) implies that $f(P, \mathcal{D}_{upd-}) = f(P, \mathcal{D}_{org})$ and thus $sup(P, \mathcal{D}_{upd-}) = sup(P, \mathcal{D}_{org})$. 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, $[P]_{\mathcal{D}_{upd-}} \in \mathcal{F}(\mathcal{D}_{upd-}, ms_{\%})$.

Next we prove $Clo([P]_{\mathcal{D}_{upd-}}) = Clo([P]_{\mathcal{D}_{org}})$. Let $C = Clo([P]_{\mathcal{D}_{org}})$ 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 4.4, we know $C' \notin [P]_{\mathcal{D}_{org}}$ and $f(C', \mathcal{D}_{org}) \neq f(P, \mathcal{D}_{org})$. Also since $P \notin \mathcal{D}_{dec}$, $C \notin \mathcal{D}_{dec}$ ($C \in [P]_{\mathcal{D}_{org}}$) and $C' \notin \mathcal{D}_{dec}$ ($C' \supset C$). Thus $f(C', \mathcal{D}_{dec}) = \emptyset$. Therefore, $f(C', \mathcal{D}_{upd-}) = f(C', \mathcal{D}_{org}) - f(C', \mathcal{D}_{dec}) = f(C', \mathcal{D}_{org}) - \emptyset = f(C', \mathcal{D}_{org})$. Combining the facts that $f(C', \mathcal{D}_{org}) \neq f(P, \mathcal{D}_{org})$ and $f(P, \mathcal{D}_{org}) = f(P, \mathcal{D}_{upd-})$, 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 4.5, C is the closed pattern of $[P]_{\mathcal{D}_{upd-}}$.

Then we prove $Keys([P]_{\mathcal{D}_{upd-}}) = min\{K | K \in Keys([P]_{\mathcal{D}_{org}}) \vee K \in Keys([Q_i]_{\mathcal{D}_{org}}), 1 \leq i \leq n\}$. This formula states that the generators of the equivalence class $[P]_{\mathcal{D}_{upd-}}$ are the set of minimum (equivalent to the most general) generators in the merging equivalence classes. This basically follows from the definition of generators in Definition 4.4.

Scenario 5 is complement of Scenario 2. Therefore, it can be proven in the same way as Scenario 2.

Last we prove that the theorem is complete. For patterns $P \notin \mathcal{D}_{dec}$, it is obvious that Scenario 1 and 2 enumerated all possible cases. For patterns $P \in \mathcal{D}_{dec}$, it is also obvious that Scenario 3 to 5 enumerated all possible cases. Therefore, the theorem is complete and correct.

□

Theorem 6.4 *PSM- presented in Algorithm 4 correctly maintains the frequent pattern space, which is represented using equivalence classes, for decremental updates.*

Proof: *According to Theorem 6.2, after an decremental update, 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) $P \notin \mathcal{D}_{dec}$ *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 for all transactions in* $\mathcal{D}_{dec}$*. Thus, Line 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 6.3). As a result,* $[P]_{\mathcal{D}_{org}}$ *is included in* $\mathcal{F}_{upd-}$ *unchanged in Line 29 as desired.*

For Scenario 2, suppose (i) $P \notin \mathcal{D}_{dec}$ *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 Line 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 other equivalence classes* $EC_1, \cdots, EC_n$ *that satisfy Line 21 and will merge with* $[P]_{\mathcal{D}_{org}}$*. When the for-loop between Line 19 to 28 completes, all these equivalence classes* $EC_1, \cdots, EC_n$ *will merge 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) $P \in \mathcal{D}_{dec}$ *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) $P \in \mathcal{D}_{dec}$*, (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 for some transactions in* $\mathcal{D}_{dec}$*. 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 Line 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 6.3). 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 an 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.* □