Chapter XIV
Maintenance of Frequent Patterns: A Survey

Mengling Feng
Nanyang Technological University, Singapore

Jinyan Li
Nanyang Technological University, Singapore

Guozhu Dong
Wright State University, USA

Limsoon Wong
National University of Singapore, Singapore

ABSTRACT

This chapter surveys the maintenance of frequent patterns in transaction datasets. It is written to be accessible to researchers familiar with the field of frequent pattern mining. The frequent pattern maintenance problem is summarized with a study on how the space of frequent patterns evolves in response to data updates. This chapter focuses on incremental and decremental maintenance. Four major types of maintenance algorithms are studied: Apriori-based, partition-based, prefix-tree-based, and concise-representation-based algorithms. The authors study the advantages and limitations of these algorithms from both the theoretical and experimental perspectives. Possible solutions to certain limitations are also proposed. In addition, some potential research opportunities and emerging trends in frequent pattern maintenance are also discussed[1].

INTRODUCTION

A frequent pattern, also named as a frequent itemset, refers to a pattern that appears frequently in a particular dataset. The concept of frequent pattern is first introduced in Agrawal et al. (1993). Frequent patterns play an essential role in various knowledge discovery and data mining (KDD) tasks, such as the discovery of association rules (Agrawal et al. 1993), correlations (Brin et al.1997), causality (Silverstein et al. 1998), sequential patterns (Agrawal et al. 1995), partial periodicity (Han et al. 1999), emerging patterns (Dong & Li 1999), etc.

Updates are a fundamental aspect of data management in frequent pattern mining applications. Other than real-life updates, they are also used in interactive data mining to gauge the impact caused by hypothetical changes to the data. When a database is updated frequently, repeating the knowledge discovery process from scratch during each update causes significant computational and I/O overheads. Therefore, it is important to analyse how the discovered knowledge may change in response to updates, so as to formulate more effective algorithms to maintain the discovered knowledge on the updated database.

This chapter studies the problem of frequent pattern maintenance and surveys some of the current work. We give an overview of the challenges in frequent pattern maintenance and introduce some specific approaches that address these challenges. This should not be taken as an exhaustive account as there are too many existing approaches to be included.

The current frequent pattern maintenance approaches can be classified into four main categories: 1) *Apriori*-based approaches, 2) *Partition*-based approaches, 3) *Prefix-tree*-based approaches and 4) *Concise-representation*-based approaches. In the following section, the basic definitions and concepts of frequent pattern maintenance are introduced. Next, we study some representative frequent pattern maintenance approaches from both theoretical and experimental perspectives. Some potential research opportunities and emerging trends in frequent pattern maintenance are also discussed.

PRELIMINARIES AND PROBLEM DESCRIPTION

Discovery of Frequent Patterns

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of distinct literals called 'items'. A 'pattern', or an 'itemset', is a set of items. A 'transaction' is a non-empty set of items. A 'dataset' is a non-empty set of transactions. A pattern P is said to be contained or included in a transaction T if $P \subseteq T$. A pattern P is said to be contained in a dataset D, denoted as $P \in D$, if there is $T \in D$ such that $P \subseteq T$. The 'support count' of a pattern P in a dataset D, denoted $count(P,D)$, is the number of transactions in D that contain P. The 'support' of a pattern P in a dataset D, denoted $sup(P,D)$, is calculated as $sup(P,D) = count(P,D)/|D|$. Figure 1(a) shows a sample dataset, and all the patterns contained in the sample dataset are enumerated in Figure 1(b) with their support counts.

A pattern P is said to be *frequent* in a dataset D if $sup(P,D)$ is greater than or equal to a prespecified threshold $ms_{\%}$. Given a dataset D and a support threshold $ms_{\%}$, the collection of all frequent itemsets in D is called the 'space of frequent patterns', and is denoted by $F(ms_{\%}, D)$. The task of frequent pattern mining is to discover all the patterns in the space of frequent patterns. In real-life applications, the size of the frequent pattern space is often tremendous. According to the definition, suppose the dataset has l distinct items, the size of the frequent pattern space can go up to 2^l. To increase computational efficiency and reduce memory usage, concise representations are developed to summarize the frequent pattern space.

Concise Representations of Frequent Patterns

The concise representations of frequent patterns are developed based on the *a priori* (or anti-monotone) property (Agrawal et al. 1993) of frequent patterns.

FACT 1 (A priori Property). Given a dataset D and a support threshold $ms_{\%}$, if pattern $P \in F(D,ms_{\%})$, then for every pattern $Q \subseteq P$, $Q \in F(D,ms_{\%})$; on the other hand, if pattern $P \notin F(D,ms_{\%})$, then for every pattern $Q \supseteq P$, $Q \notin F(D,ms_{\%})$.

The *a priori* property basically says that all subsets of frequent patterns are frequent and all supersets of infrequent patterns are infrequent.

The commonly used concise representations of frequent patterns include maximal patterns (Bayardo 1998), closed patterns (Pasquier et al.1999), key patterns (a.k.a. generators) (Pasquier et al. 1999) and equivalence classes (Li et al. 2005). Figure 1(b) graphically demonstrates how the frequent pattern space of the sample dataset can be concisely summarized with maximal patterns, closed patterns and key patterns, and Figure 1(c) illustrates how the pattern space can be compactly represented with equivalence classes.

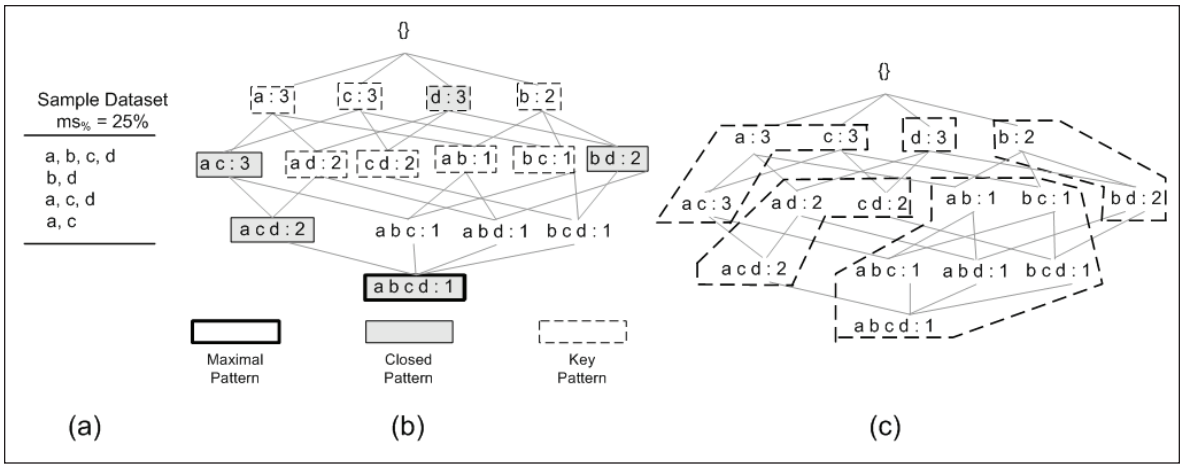
Maximal Pattern Representation

Maximal patterns are first introduced in Bayardo (1998). Frequent maximal patterns refer to the longest patterns that are frequent, and they are formally defined as follows.

Definition 1 (Maximal Pattern). Given a dataset D and a support threshold $ms_{\%}$, a pattern P is a frequent 'maximal pattern', iff $sup(P,D) \geq ms_{\%}$ and, for every $Q \supset P$, it is the case that $sup(Q,D) < ms_{\%}$.

The maximal pattern representation is composed of a set of frequent maximal patterns annotated with their support values. The maximal pattern representation is the most compact representation of the frequent pattern space. As shown in Figure 1(b), one maximal pattern is already sufficient to represent the entire pattern space that consists of 15 patterns. Based on the *a priori* property (Agrawal et al. 1993) of frequent

Figure 1. (a) An example of transaction dataset. (b) The space of frequent patterns for the sample dataset in (a) when $ms_{\%}$=25% and the concise representations of the space. (c) Decomposition of frequent pattern space into equivalence classes.



patterns, one can enumerate all frequent patterns from the frequent maximal patterns. However, the representation lacks the information to derive the exact support of frequent patterns. Therefore, the maximal pattern representation is a lossy representation.

Closed Pattern and Key Pattern Representations

Unlike the maximal pattern representation, both the closed pattern and key pattern representations are lossless concise representations of frequent patterns. We say a representation is lossless if it is sufficient to derive and determine the support of all frequent patterns without accessing the datasets. The concepts of closed patterns and key patterns are introduced together in Pasquier (1999).

Definition 2 (Closed Pattern). Given a dataset D, a pattern P is a 'closed pattern', iff for every $Q \supset P$, it is the case that $sup(Q,D) < sup(P,D)$.

For a dataset D and support threshold $ms_{\%}$, the closed pattern representation is constructed with the set of frequent closed patterns, denoted as $FC(D,ms_{\%})$, and their corresponding support information. Algorithms, such as *FPclose* (Grahne et al. 2003), *CLOSET* (Pei et al 2000) & *CLOSET+* (Wang et al 2003), have been proposed to generate the closed pattern representation effectively. As shown in Figure 1(b), the closed pattern representation is not as compact as the maximal representation. However, it is a lossless representation. The closed pattern representation can enumerate as well as derive the support values of all frequent patterns. For any frequent pattern P in dataset D, its support can be calculated as: $sup(P,D) = max\{sup(C,D) | C \supseteq P,\ C \in FC(D,ms_{\%})\}$.

Definition 3 (Key Pattern). Given a dataset D, a pattern P is a 'key pattern', iff for every $Q \subset P$, it is the case that $sup(Q,D) > sup(P,D)$.

For a dataset D and support threshold $ms_{\%}$, the key pattern representation is constructed with the set of frequent key patterns, denoted as $FG(D,ms_{\%})$, and their corresponding support information. The key pattern representation is also lossless. For any frequent pattern P in dataset D, its support can be calculated as: $sup(P,D) = min\{sup(G,D) | G \subseteq P,\ G \in FG(D,ms_{\%})\}$.

Equivalence Class Representation

Li et al. (2005) have discovered that the frequent pattern space can be structurally decomposed into sub-spaces --- equivalence classes.

Definition 4 (Equivalence Class). Let the 'filter', $f(P,D)$, of a pattern P in a dataset D be defined as $f(P,D) = \{T \in D \mid P \subseteq T\}$. Then the 'equivalence class' $[P]_D$ of P in a dataset D is the collection of patterns defined as $[P]_D = \{Q \mid f(P,D) = f(Q,D),\ Q \text{ is a pattern in } D\}$.

In other words, two patterns are 'equivalent' in the context of a dataset D iff they are included in exactly the same transactions in D. Thus the patterns in a given equivalence class have the same support. Figure 1(c) graphically illustrates how the frequent pattern space of the sample dataset can be decomposed and summarized into 5 equivalence classes. As shown in Figure 1, concise representations provide us effective means to compress the space of frequent patterns. Concise representations not only help to save memory spaces, but, more importantly, they greatly reduce the size of the ~~searching~~ space and thus the complexity of the discovery and maintenance problems of frequent patterns.

MAINTENANCE OF FREQUENT PATTERNS

Data is dynamic in nature. Datasets are often updated in the applications of frequent pattern mining. Data update operations include addition/removal of items, insertion/deletion of transactions, modifications of existing transactions, etc. In this chapter, we focus on two most common

update scenarios, where new transactions are inserted into the original dataset and obsolete transactions are removed.

When new transactions are added to the original dataset, the new transactions are called the 'incremental dataset', and the update operation is called the 'incremental update'. The associated maintenance process is called the 'incremental maintenance'. When obsolete transactions are removed from the original dataset, the removed transactions are called the 'decremental dataset', and the update operation is called the 'decremental update'. The associated maintenance process is called the 'decremental maintenance'. For the rest of the chapter, we use notations D_{org} to denote the original dataset, D_{upd} to denote the updated dataset, d^+ to denote the incremental dataset and d^- to denote the decremental dataset. In incremental updates, where new transactions d^+ are added, we have $D_{upd} = D_{org} \cup d^+$ and thus $|D_{upd}| = |D_{org}| + |d^+|$. On the other hand, in decremental updates, where existing transactions are removed, we have $D_{upd} = D_{org} - d^-$ and thus $|D_{upd}| = |D_{org}| - |d^-|$.

To effectively maintain the space of frequent patterns, we first need to understand how the space evolves in the response to data updates. Suppose we have a dataset D_{org} and the corresponding frequent pattern space $F(ms_{\%}, D_{org})$ under support threshold $ms_{\%}$. We can characterize the evolution of the frequent pattern space by studying the behaviour of individual patterns. In incremental updates, we observe that, for every pattern P, exact one of the following 4 scenarios holds:

1. $P \notin F(ms_{\%}, D_{org})$ and P is not in d^+. This corresponds to the scenario where pattern P is infrequent in D_{org} and it is not contained in d^+. In this case, pattern P remains infrequent and no update action is required.
2. $P \in F(ms_{\%}, D_{org})$ and P is not in d^+. This corresponds to the scenario where pattern P is frequent in D_{org} but it is not contained in d^+. In this case, $count(P, D_{upd}) = count(P, D_{org})$, and since $|D_{upd}| = |D_{org}| + |d^+| > |D_{org}|$, $sup(P, D_{upd}) < sup(P, D_{org})$. The support count of P remains unchanged but its support decrease. Then we have two cases: first, if $count(P, D_{upd}) \geq |D_{upd}| \times ms_{\%}$, pattern P remains to be frequent, and only its support value needs to be updated; second, if $count(P, D_{upd}) < |D_{upd}| \times ms_{\%}$, pattern P becomes infrequent in D_{upd}, and it needs to be discarded.
3. $P \notin F(ms_{\%}, D_{org})$ and P is in d^+. This corresponds to the scenario where pattern P is infrequent in D_{org} but it is contained in d^+. In this case, $count(P, D_{upd}) = count(P, D_{org}) + count(P, d^+)$. Then we have two cases: first, if $count(P, D_{upd}) \geq |D_{upd}| \times ms_{\%}$, pattern P emerges to be frequent in D_{upd}, and it needs to be included in $F(ms_{\%}, D_{upd})$; second, if $count(P, D_{upd}) < |D_{upd}| \times ms_{\%}$, pattern P remains to be infrequent, and no update action is required.
4. $P \in F(ms_{\%}, D_{org})$ and P is in d^+. This corresponds to the scenario where pattern P is frequent in D_{org} and it is contained in d^+. Similar to scenario 3, $count(P, D_{upd}) = count(P, D_{org}) + count(P, d^+)$. Again we have two cases: first, if $count(P, D_{upd}) \geq |D_{upd}| \times ms_{\%}$, pattern P remains to be frequent, and only its support value needs to be updated; second, if $count(P, D_{upd}) < |D_{upd}| \times ms_{\%}$, pattern P becomes infrequent in D_{upd}, and it needs to be discarded.

For decremental updates, similar scenarios can be derived to describe the evolution of the frequent pattern space. (Detailed scenarios can be derived easily based on the duality between incremental updates and decremental updates. Thus, details are not included.) The key observation is that both incremental and decremental updates may cause existing frequent patterns to become infrequent and may induce new frequent patterns to emerge. Therefore, the major tasks and challenges in frequent pattern maintenance are to:

1. Find ~~out~~ and discard the existing frequent patterns that are no longer frequent after the update.
2. Generate the newly emerged frequent patterns.

Since the size of the frequent pattern space is usually ~~tremendous~~, effective techniques and algorithms are required to address these two tasks.

MAINTENANCE ALGORITHMS

The maintenance of frequent patterns has attracted considerable research attention in the last decade. The proposed maintenance algorithms fall into four main categories: 1) *Apriori*-based, 2) *Partition*-based, 3) *Prefix-tree*-based and 4) *Concise-representation*-based. In this section, we will study these four types of approaches first from the theoretical perspective and then proceed on to the experimental investigation of their computational effectiveness.

Apriori-Based Algorithms

Apriori (Agrawal et al. 1993) is the first frequent pattern mining algorithm. *Apriori* discovers frequent patterns iteratively. In each iteration, it generates a set of candidate frequent patterns and then verifies them by scanning the dataset. *Apriori* defines a 'candidate-generation-verification' framework for the discovery of frequent patterns. Therefore, in *Apriori* and *Apriori-based* algorithms, the major challenge is to generate the minimum number of unnecessary candidate patterns.

FUP (Cheung et al. 1996) is the representative *Apriori*-based maintenance algorithm. It is proposed to address the incremental maintenance of frequent patterns. Inspired by *Apriori*, *FUP* updates the space of frequent patterns based on the candidate-generation-verification framework. Using a different approach from *Apriori*, *FUP* makes use of the support information of the previously discovered frequent patterns to reduce the number of candidate patterns. *FUP* effectively prunes unnecessary candidate patterns based on the following two observations.

FACT 2. Given a dataset D_{org}, the incremental dataset d^+ , the updated dataset $D_{upd} = D_{org} \cup d^+$ and the support threshold $ms_{\%}$, for every pattern $P \in F(ms_{\%}, D_{org})$, if $P \notin F(ms_{\%}, D_{upd})$, then for every pattern $Q \supseteq P$, $Q \notin F(ms_{\%}, D_{upd})$.

FACT 2 is an extension of the *a priori* property of frequent patterns. It is to say that, if a previously frequent pattern becomes infrequent in the updated dataset, then all its supersets are definitely infrequent in the updated dataset and thus should not be included as candidate patterns. FACT 2 facilitates us to discard existing frequent patterns that are no longer frequent. FACT 3 then provides us a guideline to eliminate unnecessary candidates for newly emerged frequent patterns.

FACT 3. Given a dataset D_{org}, the incremental dataset d^+ , the updated dataset $D_{upd} = D_{org} \cup d^+$ and the support threshold $ms_{\%}$, for every pattern $P \notin F(ms_{\%}, D_{org})$, if $sup(P, d^+) < ms_{\%}$, $P \notin F(ms_{\%}, D_{upd})$.

FACT 3 states that, if a pattern is infrequent in both the original dataset and the incremental dataset, it is definitely infrequent in the updated dataset. This allows us to eliminate disqualified candidates of the newly emerged frequent patterns based on their support values in the incremental dataset. The support values of candidates can be obtained by scanning only the incremental dataset. This greatly reduces the number of scans of the original dataset and thus improves the effectiveness of the algorithm. (In general, the size of the incremental dataset is much smaller than the one of the original dataset.)

In Cheung et al. (1997), *FUP* is generalized to address the decremental maintenance of frequent patterns as well. The generalized version of *FUP* is called *FUP2H*. Both *FUP* and *FUP2H* generate a much smaller set of candidate patterns compared

to *Apriori*, and thus they are more effective. But both *FUP* and *FUP2H* still suffer from two major drawbacks:

1. they require multiple scans of the original and incremental/decremental datasets to obtain the support values of candidate patterns, which leads to high I/O overheads, and
2. they repeat the enumeration of previously discovered frequent patterns.

To address Point 2, Aumann et al (1999) proposed a new algorithm---*Borders*.

Borders is inspired by the concept of the 'border pattern', introduced in Mannila & Toivonen (1997). In the context of frequent patterns, the 'border pattern' is formally defined as follows.

Definition 5 (Border Pattern). Given a dataset D and minimum support threshold $ms_{\%}$, a pattern P is a 'border pattern', iff for every $Q \subset P$, $Q \in F(ms_{\%}, D)$ but $P \notin F(ms_{\%}, D)$.

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

We illustrate the idea of *Borders* using an incremental update example. When the incremental dataset d^+ is added, *Borders* first scans through d^+ to update the support values of the existing frequent patterns and the border patterns. If no border patterns emerge to be frequent after the update, the maintenance process is finished. Otherwise, if some border patterns become frequent after the update, new frequent patterns and border patterns need to be enumerated. Those border patterns that emerge to be frequent after the update are called the 'promoted border patterns'. The pattern enumeration process follows the *Apriori* candidate-generation-verification method. But, distinct from *Apriori* and *FUP*, *Borders* resumes the pattern enumeration from the 'promoted border patterns' onwards and thus avoids the enumeration of previously discovered frequent patterns.

Since *Borders* successfully avoids unnecessary enumeration of previously discovered patters, it is more effective than *FUP*. However, similar to *FUP*, *Borders* requires multiple scans of original and incremental/decremental datasets to obtain the support values of newly emerged frequent pattern and border patterns. *Borders* also suffers from heavy I/O overheads. One possible way to solve this limitation of *FUP* and *Borders* is to compress the datasets into a prefix-tree (Han et al. 2000). The prefix-tree is a data structure that compactly records datasets and thus enables us to obtain support information of patterns without scanning of the datasets. Details will be discussed in the section of Prefix-tree-based algorithms.

Partition-Based Algorithms

Partition-based maintenance algorithms, similar to *Apriori*, enumerate frequent patterns based on the candidate-generation-verification framework, but they generate candidate patterns in a different manner. Candidate patterns are generated based on the 'partition-based heuristic' (Lee et al. 2005): given a dataset D that is divided into n partitions $p_1, p_2, \ldots, p_n$, if a pattern P is a frequent pattern in D, then P must be frequent in at least one of the n partitions of D.

Sliding Window Filtering (*SWF*) (Lee et al. 2005) is a recently proposed partition-based algorithm for frequent pattern maintenance. *SWF* focuses on the pattern maintenance of time-variant datasets. In time-variant datasets, data updates involve both the insertion of the most recent transactions (incremental update) and the deletion of the most obsolete transactions (decremental update).

Given a time-variant dataset D, *SWF* first divides D into n partitions and processes one partition at a time. The processing of each partition is

called a *phase*. In each *phase*, the local frequent patterns are discovered, and they are carried over to the next *phase* as candidate patterns. In this manner, candidate patterns are cumulated progressively over the entire dataset *D*. The set of cumulated candidate patterns is called the 'cumulative filter', denoted by *CF*. According to the 'partition-based heuristic', *CF* is the superset of the set of frequent patterns. Finally, *SWF* scans through the entire dataset to calculate the actual support of the candidate patterns and to decide whether they are globally frequent. To facilitate the maintenance of frequent patterns, *SWF* records not only the support information but also the 'start partition' of each candidate pattern. The 'start partition' attribute of candidate patterns refers to the first partition that the candidate pattern is first introduced. When the most obsolete transactions are removed, the 'start partition' attribute allows us to easily locate and thus update the candidate patterns that are involved in the obsolete transactions. When new transactions are added, the incremental dataset d^+ will be treated as a partition of the dataset and will be involved in the progressively generation of candidate patterns.

The major advantage of *SWF* is that, based on the 'partition-based heuristic', *SWF* prunes most of the false candidate patterns in the early stage of the maintenance process. This greatly reduces the computational and memory overhead. Moreover, *SWF* requires only one scan of the entire time-variant dataset to verify the set of candidate patterns. We will demonstrate in our experimental studies later that it is this very advantage of *SWF* that allows it to significantly outperform *Apriori* and *FUP*.

PREFIX-TREE-BASED ALGORITHMS

The prefix-tree is an effective data structure that compactly represents the transactions and thus the frequent patterns in datasets. The usage of the prefix-tree is a tremendous breakthrough in frequent pattern discovery. With the prefix-tree, we can compress the transactional dataset and store it in the main memory. This enables fast access of the support information of all the frequent patterns. More importantly, we can now generate frequent patterns by traversing the prefix-tree without multiple scanning of the dataset and generation of any candidate patterns (Han et al. 2000). To better appreciate the idea of prefix-tree, let us study the *FP-tree*, the most commonly used prefix-tree, as an example.

The *FP-tree*, in full 'frequent pattern tree', is first proposed in Han et al. (2000). The *FP-tree* is a compact representation of all relevant fre-

Figure 2. (a) The original dataset. (b) The projected dataset from the original dataset. (c) The construction process of FP-tree.

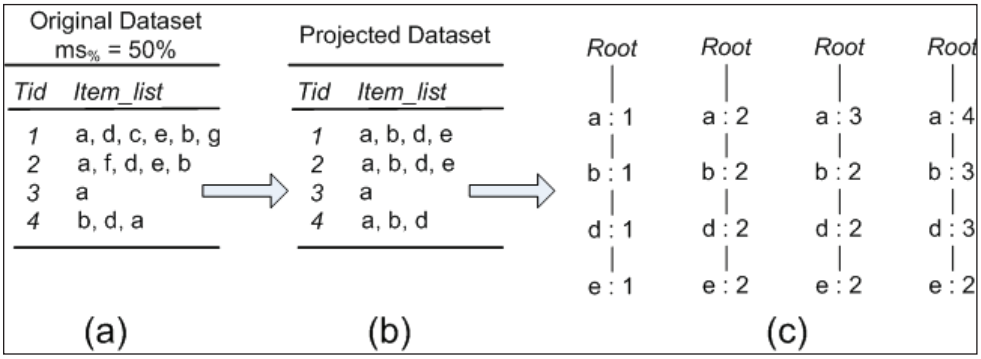
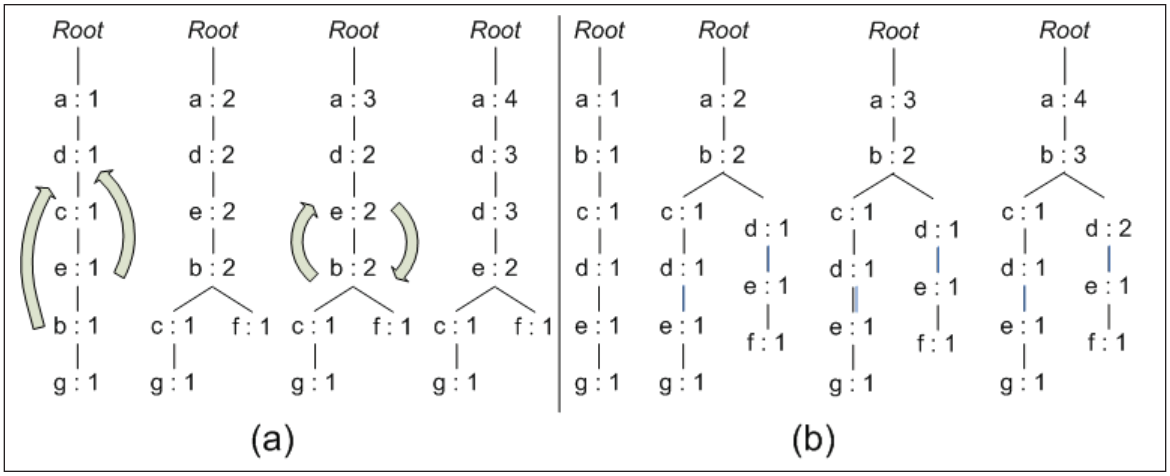


Figure 3. (a) The construction of CATS tree. (b) The construction of CanTree.



quency information in a database. Every branch of the *FP-tree* represents a 'projected transaction' and also a candidate pattern. The nodes along the branches are stored in decreasing order of support values of the corresponding items, so leaves are representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping transactions share prefixes of the corresponding branches. Figure 2 demonstrates how the *FP-tree* is constructed for the sample dataset given a support threshold $ms_{\%}$. First, the dataset is transformed into the 'projected dataset'. In the 'projected dataset', all the infrequent items are removed, and items in each transaction are sorted in descending order of their support values. Transactions in the 'projected dataset' are named the 'projected transactions'. The 'projected transactions' are then inserted into the prefix-tree structure one by one, as shown in Figure 2(c). It can be seen that the *FP-tree* effectively represents the sample dataset in Figure 2(a) with only four nodes.

Based on the idea of *FP-tree*, a novel frequent pattern discovery algorithm, known as *FP-growth*, is proposed. *FP-growth* generates frequent pattern by traversing the *FP-tree* in a depth-first manner. *FP-growth* only requires two scans of the dataset to construct the *FP-tree* and no candidate generations. (The detailed frequent pattern generation process can be referred to Han et al. (2000)). *FP-growth* is a very effective algorithm. It is experimentally shown that *FP-growth* can outperform *Apriori* by orders of magnitudes.

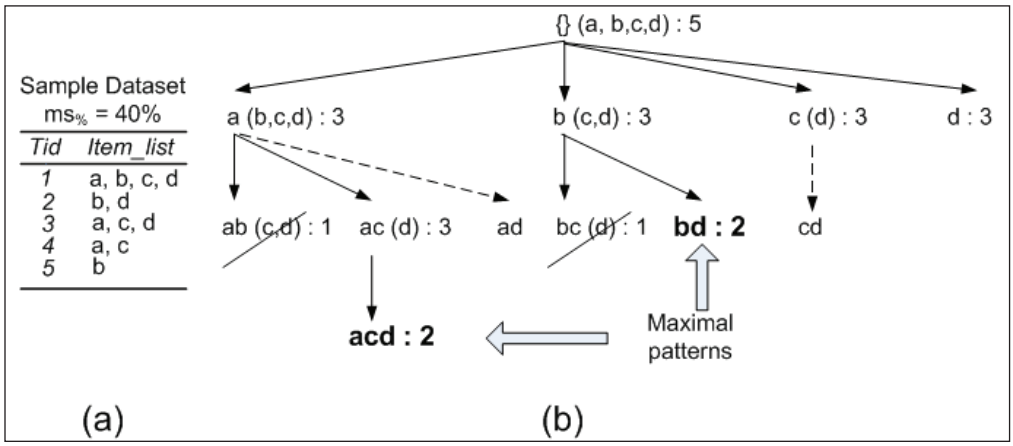
Now the question is, when the dataset is updated, how to effectively update the prefix-tree and thus to achieve efficient maintenance of frequent patterns? To answer this question, Koh & Shieh (2004) developed the *AFPIM* (Adjusting FP-tree for Incremental Mining) algorithm *AFPIM*, as the name suggested, focuses on the incremental maintenance of frequent patterns. *AFPIM* aims to update the previously constructed *FP-tree* by scanning only the incremental dataset. Recall that, in *FP-tree*, frequent items are arranged in descending order of their support values. Insertions transactions may affect the support values and thus the ordering of items in the *FP-tree*. When the ordering is changed, items in the *FP-tree* need to be adjusted. In *AFPIM*, this adjustment is accomplished by re-sorting the items through bubble sort. Bubble sort sorts items by recursively

exchanging adjacent items. This sorting process is computational expensive, especially when the ordering of items are dramatically affected by the data updates. In addition, incremental update may induce new frequent items to emerge. In this case, the *FP-tree* can no longer ~~by~~ adjusted using *AFPIM*. Instead, *AFPIM* has to scan the updated dataset to construct a new *FP-tree*.

To address the limitations of *AFPIM*, Cheung & Zaïane (2003) proposed the *CATS tree* (Compressed and Arranged Transaction Sequences tree), a novel prefix-tree for frequent patterns. Compared to the *FP-tree*, the *CATS tree* introduces a few new features. First, the *CATS tree* stores all the items in the transactions, regardless whether the items are frequent or not. This feature of *CATS tree* allows us to update *CATS tree* even when new frequent items have emerged. Second, to achieve high compactness, *CATS tree* arranges nodes based on their local support values. Figure 3(a) illustrates how the *CATS tree* of the sample dataset in Figure 2(a) is constructed and how the nodes in the tree are locally sorted. In the case of incremental updates, the *CATS tree* is updated by merging the newly inserted transactions with the existing tree branches. According to the construction method of *CATS tree*, transactions in incremental datasets can only be merged into the *CATS tree* one by one. Moreover, for each new transaction, searching though the *CATS tree* is required to find the right path for the new transaction to merge in. In addition, since nodes in *CATS tree* are locally sorted, swapping and merging of nodes are required during the update of the *CATS tree* (as shown in Figure 3(a)).

CanTree (Leung et al. 2007), Canonical-order Tree, is another prefix-tree designed for the maintenance of frequent patterns. The *CanTree* is constructed in a similar manner as the *CATS tree*, as shown in Figure 3(b). But in the *CanTree*, items are arranged according to some canonical order, which can be determined by the user prior to the mining process. For example, items in the *CanTree* can be arranged in lexicographic order, or, alternatively, items can be arranged based on certain property values of items (e.g. their prices, their priority values, etc.). Note that, in *CanTree*, once the ordering of items is fixed, items will follow this ordering for all the subsequent updates. To handle data updates, the *CanTree* allows new transactions to be inserted easily. Unlike the *CATS tree*, transaction insertions in the *CanTree* require

Figure 4. (a) Sample dataset. (b) The backtracking tree of the sample dataset when $ms_{\%}$=40%. Bolded nodes are the frequent maximal patterns, nodes that are crossed out are enumeration termination points, and nodes that are linked with a dotted arrow are skipped candidates.



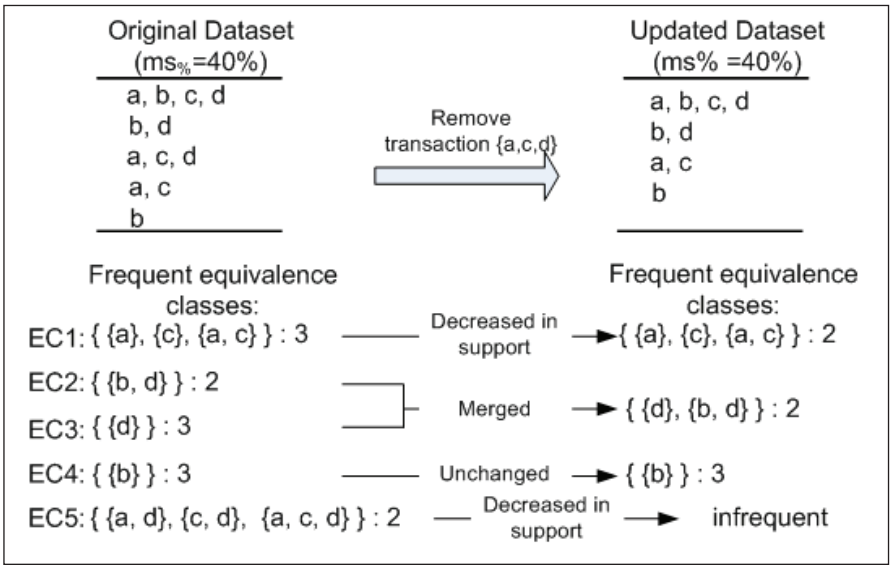
no extensive searching for merge-able paths. Also since the canonical order is fixed, any changes in the support values of items caused by data updates have no effect on the ordering of items in the *CanTree*. As a result, swapping/merging nodes are not required in the update of *CanTree*. The simplicity of the *CanTree* makes it a very powerful prefix-tree structure for frequent pattern maintenance. Therefore, in our experimental studies, we choose *CanTree* to represent the prefix-tree-based maintenance algorithms.

Concise-Representation-Based Algorithms

It is well known that the size of the frequent pattern space is usually large. The tremendous size of frequent patterns greatly limits the effectiveness of the maintenance process. To break this bottleneck, algorithms are proposed to maintain the concise representations of frequent patterns, instead of the entire pattern space. We name this type of maintenance algorithms as the concise-representation-based algorithms.

ZIGZAG (Veloso et al. 2002) and *TRUM* (Feng et al. 2007) are two representative examples of this type of algorithms. *ZIGZAG* (Veloso et al. 2002) maintains only the maximal frequent patterns. *ZIGZAG* updates the maximal frequent patterns with a backtracking search, which is guided by the outcomes of the previous mining iterations. The backtracking search method in *ZIGZAG* is inspired by its related work *GenMax* (Guoda 2001). *ZIGZAG* conceptually enumerates the candidates of maximal frequent patterns with a 'backtracking tree'. Figure 4(b) shows an example of backtracking tree. In the backtracking tree, each node is associated with a frequent pattern and its 'combination set'. For a particular frequent pattern *P*, the 'combination set' refers to the set of items that form potential candidates by combining with *P*. Take the backtracking tree in Figure 4(b) as an example. Node {*a*} is associated with combination set {*b, c, d*}. This implies that the union of {*a*} and the items in the combination set, which are {*a, b*}, {*a, c*} and {*a, d*}, are potential candidates for maximal frequent patterns.

Figure 5. The evolution of frequent equivalence classes under decremental updates.



ZIGZAG also employs certain pruning techniques to reduce the number of generated false candidates. First, *ZIGZAG* prunes false candidates based on the *a priori* property of frequent patterns. If a node in the backtracking tree is not frequent, then all the children of the node are not frequent, and thus candidate enumeration of the current branch can be terminated. In Figure 4(b), crossed out nodes are the enumeration termination points that fall in this scenario. Second, *ZIGZAG* further eliminates false candidates based on the following fact.

FACT 4. Given a dataset D and a support threshold $ms_{\%}$, if a pattern P is a maximal frequent pattern, then for every pattern $Q \supset P$, Q is not a maximal frequent pattern.

FACT 4 follows the definition of the maximal frequent pattern. In Figure 4(b), nodes, which are pointed with a dotted line, are those pruned based on this criterion.

On the other hand, *TRUM* (Transaction Removal Update Maintainer) maintains the equivalence classes of frequent patterns. *TRUM* focuses on the decremental maintenance. In Feng et al. (2007), it is discovered that, in response to decremental updates, an existing frequent equivalence class can evolve in exactly three ways as shown in Figure 5. The first way is to remain unchanged without any change in support. The second way is to remain unchanged but with a decreased support. If the support of an existing frequent equivalence class drops below the minimum support threshold, the equivalence class will be removed. The third way is to grow by merging with other classes. As a result, the decremental maintenance of frequent equivalence classes can be summarized into two tasks. The first task is to update the support values of existing frequent equivalence classes. The second task is to merge equivalence classes that are to be joined together.

Figure 6. (a) The original dataset and the frequent equivalence classes in the original dataset when $ms_{\%}$=40%. (b) The Tid-tree for the original dataset. (c) The update of the Tid-tree under decremental update.

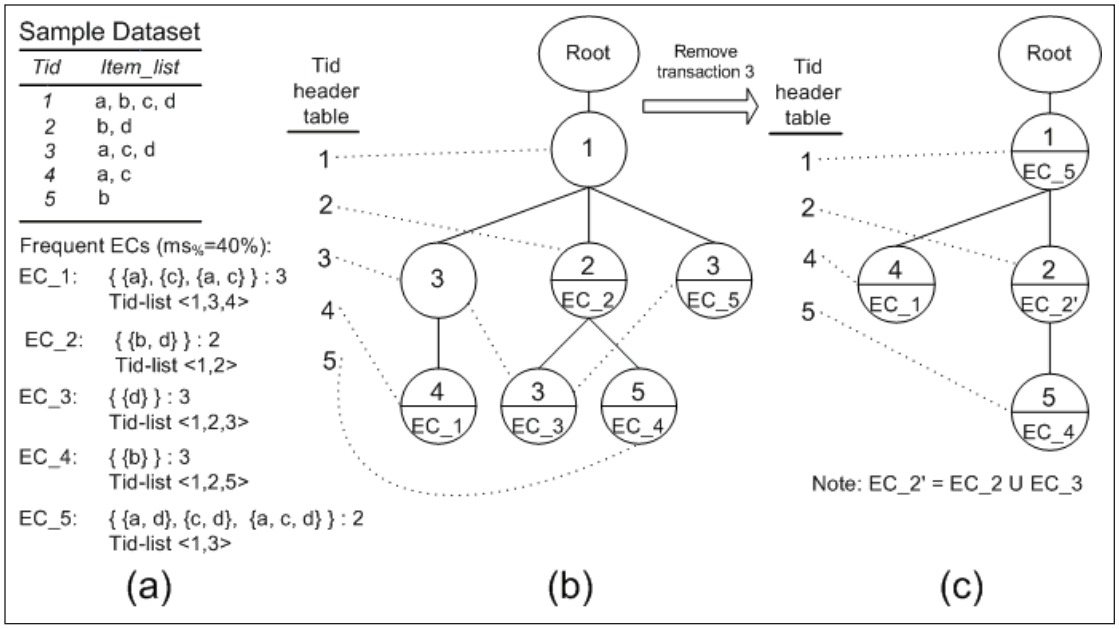


Table 1 Summary of various maintenance algorithms.

	Algorithm	Strengths	Weaknesses
Apriori-based	**FUP**	● Makes use of the support information of the previously discovered frequent patterns to reduce the number of candidate patterns	● Generates large amount of unnecessary candidates ● Requires multiple scans of datasets
	Borders	● Avoids enumeration of previous discovered patterns ● Effective enumeration of new frequent patterns from the border patterns	● Generates large amount of unnecessary candidates ● Requires multiple scans of datasets
Partition-based	**SWF**	● Prunes most of the false candidates in the early stage based on the 'partition-based heuristic' ● Requires only one full scan of dataset	● Still generates unnecessary candidates
Prefix-tree-based	**AFPIM**	● Dataset is summarized into a prefix-tree and requires only two scans of the dataset ● No false candidate is enumerated	● Inefficient update of the prefix-tree: the whole tree needs to be re-organized for each update ● The prefix-tree needs to be rebuild if new frequent items emerge
	CATS tree	● Dataset is summarized into a prefix-tree and requires only two scans of the dataset ● No false candidate is enumerated ● Items are locally sorted, which allows the tree to be locally updated ● The tree update mechanism allows new frequent items to emerge	● Node swapping and merging, which are computational expensive, are required for the local update of prefix-tree
	CanTree	● Dataset is summarized into a prefix-tree and requires only two scans of the dataset ● No false candidate is enumerated ● Items are arranged in a canonical-order that will not be affected by the data update, so that no re-sorting, node swapping and node merging are needed while updating the prefix tree	● CanTree is less compact compared to CATS tree
Concise-representation-based	**ZIGZAG**	● Updates the maximal frequent patterns with a backtracking search ● Prunes infrequent and non-maximal patterns in the early stage	● Maximal patterns are lossy representations of frequent patterns
	TRUM	● Maintains frequent patterns based on the concept of equivalence class --- a lossless representation of frequent patterns ● Employs an efficient data structure *Tid-tree* to facilitate the maintenance process	● Handles only the decremental maintenance

TRUM accomplishes the two maintenance tasks effectively with a novel data structure called the *Tid-tree*. The *Tid-tree* is developed based on the concept of Transaction Identifier List, in short *Tid-list*. Tid-lists, serve as the vertical projections of items, greatly facilitate the discovery of frequent itemsets and the calculation of their support. Moreover, *Tid-lists* can be utilized as the identifiers of equivalence classes. According to the definition of the equivalence class, each frequent equivalence class is associated with a unique *Tid-list*. The *Tid-tree* is a prefix tree of the *Tid-lists* of the frequent equivalence classes. Figure 6(b) shows how the *Tid-lists* of frequent equivalence classes in Figure 6(a) can be stored in a *Tid-tree*. The *Tid-tree* has two major features: (1) Each node in the *Tid-tree* stores a *Tid*. If the *Tid* of the node is the last *Tid* in some equivalence class's *Tid-list*, the node points to the corresponding equivalence class. Moreover, the depth of the node reflects the support of the corresponding equivalence class. (2) The *Tid-tree* has a header table, where each slot stores a linked list that connects all the nodes with the same *Tid*.

When transactions are removed from the original dataset, the *Tid-tree* can be updated by removing all the nodes corresponding to the *Tids* of the deleted transactions. This can be accomplished effectively with the help of the *Tid* header table. As demonstrated in Figure 6(c), after a node is removed, its children re-link to its parent to maintain the tree structure. If the node points to an equivalence class, the pointer is passed to its parent. When two or more equivalence class pointers collide into one node, they should be merged together. E.g. in Figure 4, equivalence classes EC 2 and EC 3 of the original dataset merge into EC 2' after the update. With the *Tid-tree*, two decremental maintenance tasks are accomplished in only one step.

We have reviewed the representative maintenance algorithms for frequent patterns. The strengths and weaknesses of these algorithms are summarized in Table 1.

Experimental Studies

We have discussed the different types of maintenance algorithms from theoretical and algorithmic perspectives. In this section, we justify our theoretical observations with experimental results. The performance of the discussed algorithms is tested using several benchmark datasets from the *FIMI* Repository, http://fimi.cs.helsinki.fi. In this chapter, the results of *T*10*I*4*D*100*K*, *mushroom, pumsb_star* and *gazelle* (a.k.a *BMS-WebView-1*) are presented. These datasets form a good representative of both synthetic and real datasets. The detailed characteristics of the datasets are presented in Table 1. The experiments were run on a PC with 2.8GHz processor and 2GB main memory.

The performance of the maintenance algorithms is investigated in two ways. First, we study their computational effectiveness over various update intervals for a fixed support threshold

Table 2. Characteristics of Datasets

Datasets	Size	#Trans	#Items	MaxTL	AvgTL
T10I4D100K	3.93MB	100,000	870	30	10.10
mushroom	0.56MB	8,124	119	23	23
pumsb_star	11.03MB	49,046	2,088	63	50.48
gazelle	0.99MB	59,602	497	268	2.51

$ms_{\%}$. For incremental updates, the update interval, denoted as Δ^+, is defined as $\Delta^+=|d^+|/|D_{org}|$. For decremental updates, the update interval, denoted as Δ^-, is defined as $\Delta^-=|d^-|/|D_{org}|$. Second, we study the computational effectiveness of the maintenance algorithms over various support thresholds $ms_{\%}$ for a fixed update interval. To better evaluate the maintenance algorithms, their performance is compared against two representative frequent pattern mining algorithms: *Apriori* (Agrawal et al. 1993) and *FP-growth* (Han et al. 2000).

First, let us look at the *Apriori*-based algorithms: the *FUP* and the *Borders*. The experimental results of *FUP* and *Borders* are summarized in Figure 7(a) and 8(a). It is discovered that both *FUP* and *Borders* outperform *Apriori* over various datasets and update intervals. *FUP* is on average around twice faster than *Apriori*, and, especially for the *mushroom* dataset, *FUP* outperforms *Apriori* up to 5 times when the update interval gets larger. Compared with *FUP*, *Borders* is much more effective. *Borders* outperforms *Apriori* on average an order of magnitude. This shows that the 'border pattern' is a useful concept that helps to avoid redundant enumeration of existing frequent patterns. However, both *FUP* and *Borders* are much slower compared to *FP-growth*, the prefix-tree based frequent pattern mining algorithm. This is mainly because both *FUP* and *Borders* require multiple scans of datasets and thus cause high I/O overhead. To solve this limitation, we employ a prefix-tree structure with the *Borders* algorithm, and we name the improved algorithm *Borders(prefixTree)*. It is experimentally demonstrate that the employment of a prefix-tree greatly improve the efficiency of *Borders*. *Borders(prefixTree)* is faster then the original *Borders* by at least an order of magnitude, and it even beats *FP-growth* in some cases.

Second, the performance results of *SWF*, the partition-based algorithm, are presented in Figure 7(b) and 8(b). *SWF* is found to be more effective than *Apriori*. *SWF* outperforms *Apriori* on average about 6 times. However, since *SWF* still follows the candidate-generation-verification framework, its performance is not as efficient as *FP-growth*, which discovers frequent patterns without generation of any candidates.

Third, we have *CanTree*[2], a prefix-tree-based algorithm. Its performance is also summarized in Figure 7(b) and 8(b). It is observed that *CanTree* is a very effective maintenance algorithm. *CanTree* is faster than both *Apriori* and *FP-growth*. It outperforms *Apriori* at least an order or magnitude. *CanTree* performs the best on the *mushroom* dataset, where it is almost 1000 times faster than *Apriori* and about 10 times faster than *FP-growth*.

Lastly, we study *ZIGZAG* and *TRUM*, which maintain the concise representations of frequent patterns. The effectiveness of *ZIGZAG* and *TRUM* is evaluated under decremental updates. They are also compared with *FUP2H*, the generalized version of *FUP*. Experimental results are summarized in Figure 7(c) and 8(c). *ZIGZAG* and *TRUM* maintains only the concise representations of frequent patterns, where the number of involved patterns is much smaller compared to the size of frequent pattern space. Therefore, they are more effective, especially for small update intervals, than the algorithms that discover or maintain frequent patterns. However, it is also observed that the advantage of *ZIGZAG* and *TRUM* diminish as the update interval increases. For some cases, *ZIGZAG* and *TRUM* are even slower than *FP-growth*. Among the comparing maintenance algorithms --- *FUP2H*, *ZIGZAG* and *TRUM*, *TRUM* is the most effective decremental maintenance algorithm.

In summary, for incremental maintenance, we found that *CanTree* is the most effective algorithm; on the other hand, for decremental maintenance, *TRUM* is the most effective one. In general, it is observed that the advantage of maintenance algorithms diminishes as the update interval increases. This is because, when more transactions are inserted/deleted, a larger number of frequent patterns are affected, and

thus a high computational cost is required to maintain the pattern space. It is inevitable that, when the update interval reaches a certain level, the frequent pattern space will be affected so dramatically that it will be better ~~off~~ to re-discover the frequent patterns than maintaining them. In addition, it is also observed that the advantage of maintenance algorithms becomes more obvious when the support threshold $ms_{\%}$ is small. It is well known that the number of frequent patterns and thus the size of the frequent pattern space grow exponentially as the support threshold drops. Therefore, when the support threshold is small, the space of frequent patterns becomes relatively large, and the discovery process becomes more 'expensive'. In this case, updating the frequent pattern space with maintenance algorithms becomes a better option.

FUTURE OPPORTUNITIES

We have reviewed the frequent pattern maintenance algorithms for conventional transaction datasets. Due to the advance in information technologies, a lot of data now is recorded continuously like a stream. This type of data is called 'data streams'.

A 'data stream' is an ordered sequence of transactions that arrives in timely order. Data streams are involved in many applications, e.g. sensor network monitoring (Halatchev & Gruenwald 2005), internet packet frequency estimation (Demaine et al. 2004), web failure analysis (Cai et al. 2004), etc. Data steams are updated constantly. Thus effective algorithms are needed for the maintenance of frequent patterns in data streams. Compared with the conventional transaction dataset, the frequent pattern maintenance in data streams is more challenging due to the following factors: first, data streams are continuous and unbounded (Leung & Khan 2006). While handling data streams, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Second, data in streams are not necessarily uniformly distributed (Leung & Khan 2006). That is to say currently infrequent patterns may emerge to be frequent in the future, and vice versa. Therefore, we can no longer simply prune out infrequent patterns. Third, updates in data streams happen more frequently and are more complicated. Data streams are usually updated in the 'sliding windows' manner, where, at each update, one obsolete transaction is removed from the window and one new transaction is added. Data streams are also updated in the 'damped' manner, in which every transaction is associated with a weight and the weight decrease with age.

The maintenance of frequent patterns in data streams faces more challenges compared to the conventional one. Some new algorithms (Manku et al 2002 & Metwally et al 2005) have been proposed to address the problem. However, certain existing ideas in the maintenance algorithms of transaction datasets could be useful to the maintenance in data streams, e.g. the prefix-tree (Leung & Khan 2006). In our opinion, to explore how the existing maintenance techniques can be used to benefit the frequent pattern maintenance in data streams is a potential and promising research direction.

CONCLUSION

This chapter has reviewed the maintenance of frequent patterns in transaction datasets. We focused on both incremental and decremental updates. We have investigated how the space of frequent patterns evolves in ~~the~~ response to ~~the~~ data updates. It is observed that both incremental and decremental updates may cause existing frequent patterns to become infrequent and may induce new frequent patterns to emerge. We then summarized the major tasks in frequent pattern

Figure 7. Computational performance over various update intervals. (a) The Apriori-based algorithms --- FUP, Borders and Borders(prefixTree). (b) Partition-based algorithm SWF and prefix-tree-based algorithm CanTree. (c) Concise-representation-based algorithms --- ZIGZAG and TRUM.

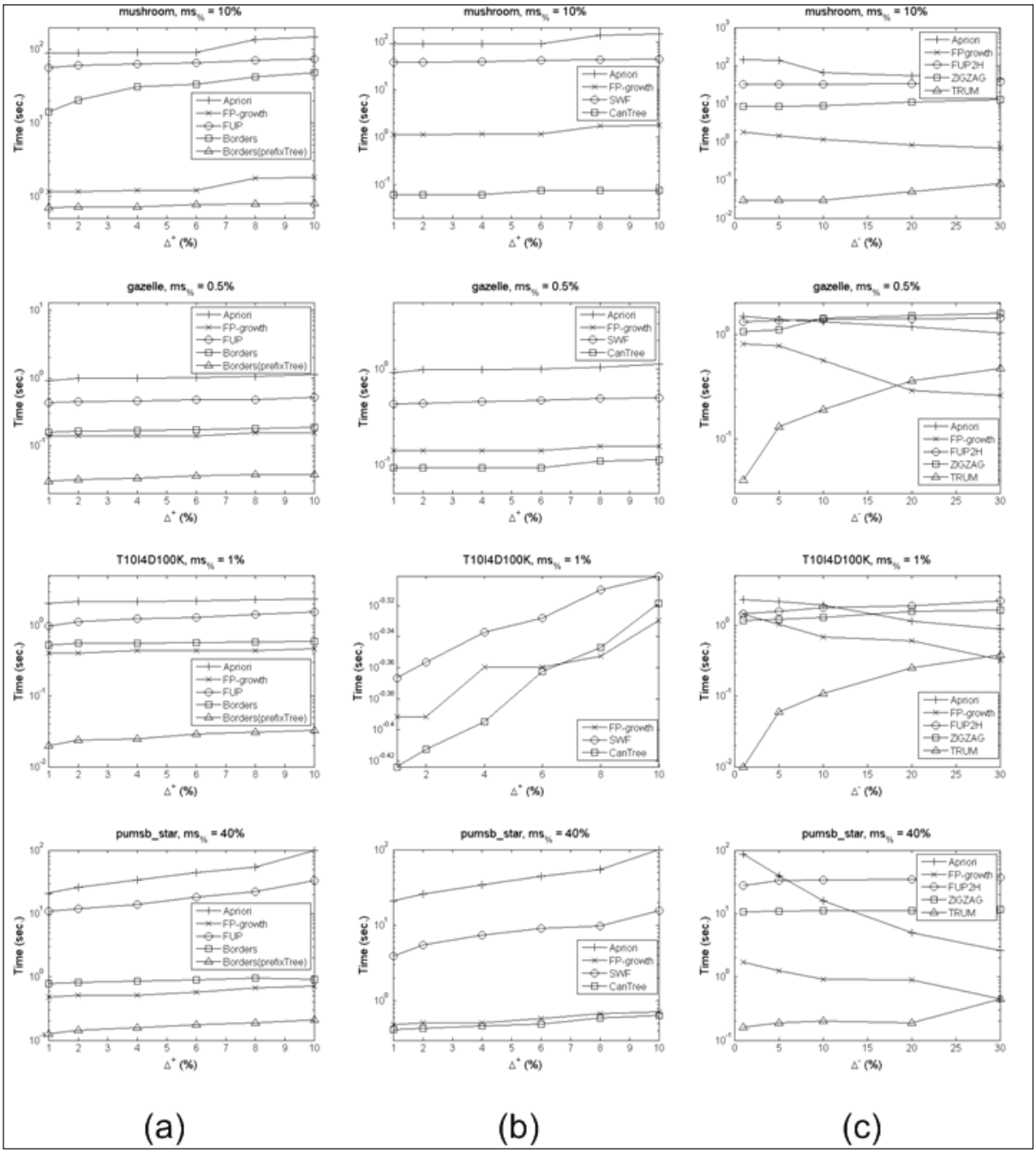
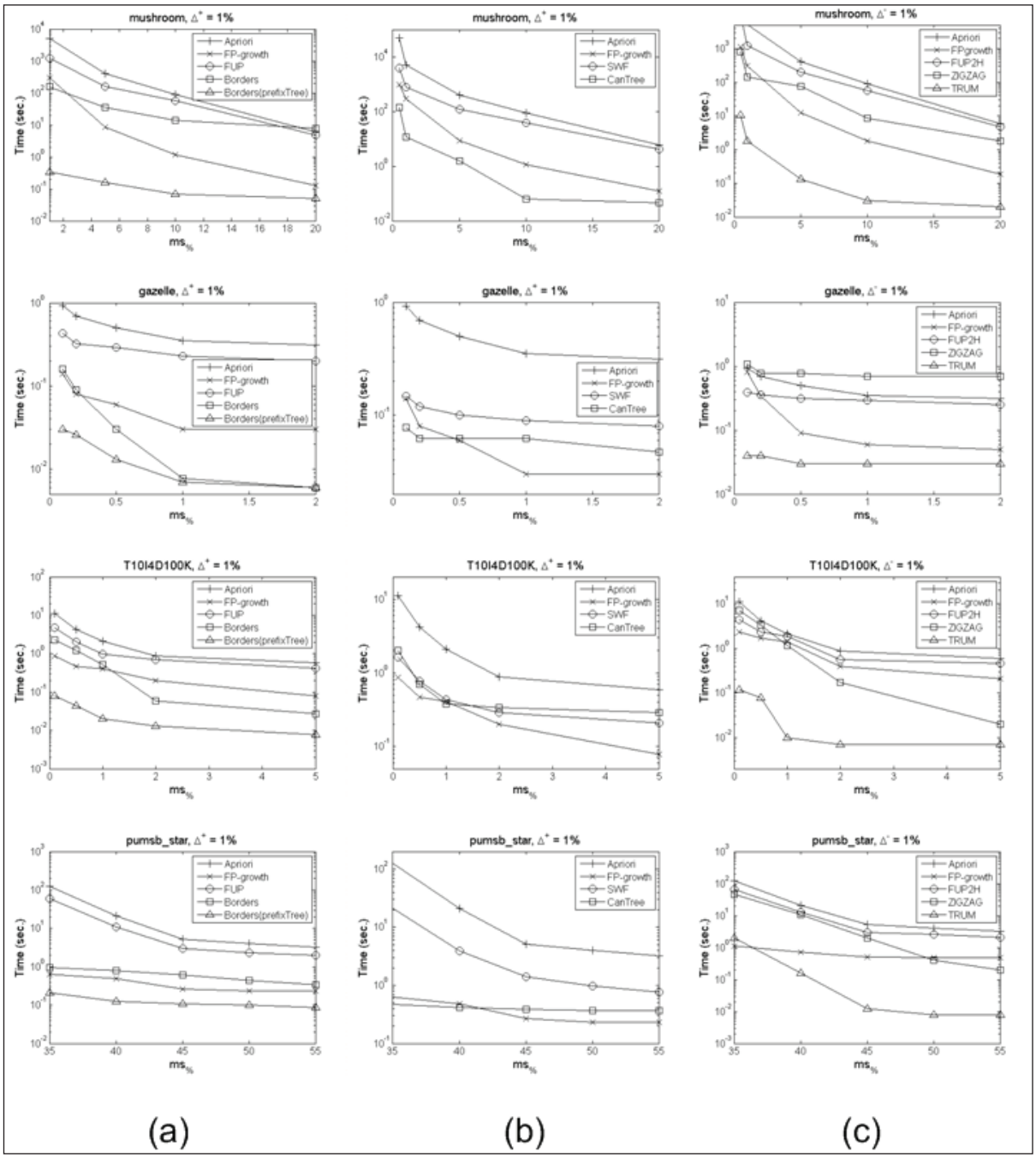


Figure 8. Computational performance over various support thresholds. (a) The Apriori-based algorithms --- FUP, Borders and Borders(prefixTree). (b) Partition-based algorithm SWF and prefix-tree-based algorithm CanTree. (c) Concise-representation-based algorithms --- ZIGZAG and TRUM.



maintenance is to 1) locate and discard previously frequent patterns that are no longer qualified and to 2) generate new frequent patterns.

We have surveyed four major types of maintenance algorithms, namely the *Apriori*-based algorithms, the partition-based algorithms, the prefix-tree-based algorithms and the concise-representation-based algorithms. The characteristics of these algorithms have been studied from both theoretical and experimental perspectives. It is observed that algorithms that involve multiple data scans suffer from high I/O overhead and thus low efficiency. We have demonstrated that this limitation can be solved by employing a prefix-tree, e.g. *FP-tree*, to summarize and store the dataset. According to the experimental studies, for incremental maintenance, the prefix-tree-based algorithm, *CanTree*, is the most effective algorithm. On the other hand, *TRUM*, which maintains the equivalence classes of frequent patterns, is the most effective method for decremental maintenance.

In addition, it is a challenging and potential research direction to explore how the existing maintenance techniques for transaction data can be applied to effectively maintain frequent patterns in data streams.

REFERENCES

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

Agrawal, R., & Srikant, R. (1995). Mining sequential patterns. *Proceedings of the Eleventh International Conference on Data Engineering* (pp. 3-14).

Aumann, Y., Feldman, R., Lipshtat, O. & Manilla, H. (1999). Borders: An efficient algorithm for association generation in dynamic databases. *Intelligent Information Systems*, *12*(1), 61-73.

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

Brin, S., Motwani, R., & Silverstein, C. (1997). Beyond market basket: Generalizing association rules to dependence rules. *Data Mining and Knowledge Discovery*, *2*(1), 39-68.

Cai, Y. D., Clutter, D., Pape, G., Han, J., Welge, M., & Auvil L. (2004). MAIDS: mining alarming incidents from data streams. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (pp. 919-920).

Cheung, D., Han, J., Ng, V. T., & Wong, C. Y. (1996). Maintenance of discovered association rules in large databases: an incremental update technique. *Proceedings of the 1996 International Conference on Data Engineering* (pp. 106-114).

Cheung, D., Lee, S. D., & Kao, B. (1997). A general incremental technique for maintaining discovered association rules. *Database Systems for Advanced Applications* (pp. 185-194).

Cheung, W., & Zaïane, O. R. (2003). Incremental mining of frequent patterns without candidate generation or support constraint. *Proceedings of the 2003 International Database Engineering and Applications Symposium* (pp. 111-116).

Demaine, E. D., López-Ortiz, A., & Munro, J. I. (2002). Frequency estimation of internet packet streams with limited space. *Proceedings of the 10th Annual European Symposium on Algorithms* (pp. 348-360).

Dong, G., & Li, J. (1999). Efficient Mining of Emerging Patterns: Discovering Trends and Differences. *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 15-18).

Feng, M., Dong, G., Li, J., Tan, Y-P., & Wong, L. (2007). Evolution and maintenance of frequent pattern space when transactions are removed. *Proceedings of the 2007 Pacific-Asia Conference on Knowledge Discovery and Data Mining* (pp. 489-497).

Grahne, G., & Zhu, J. (2003). Efficiently using prefix-trees in mining frequent itemsets. *Proceedings 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

Guoda, K., & Zaki, M. J. (2001). Efficiently mining maximal frequent itemsets. *Proceedings of the 2001 IEEE International Conference on Data Mining* (pp. 163-170).

Han, J., Dong, G., & Yin, Y. (1999). Efficient mining of partial periodic patterns in time series database. *International Conference on Data Engineering*, (pp. 106-115).

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

Halatchev, M., & Gruenwald, L. (2005). Estimating missing values in related sensor data streams. *International Conference on Management of Data* (pp. 83-94).

Jiang, N., & Gruenwald, L. (2006). Research issues in data stream association rule mining. *ACM SIGMOD Record*, *35*(1), 14-19.

Koh, J-L., & Shieh, S-F. (2004). An efficient approach for maintaining association rules based on adjusting FP-tree structures. *Proceedings of the 2004 Database Systems for Advanced Applications* (pp. 417-424).

Lee, C-H., Lin, C-R., & 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. (2006). DSTree: A Tree Structure for the Mining of Frequent Sets from Data Streams. *Proceedings of the Sixth International Conference on Data Mining* (pp. 928 - 932).

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

Li, H., Li, J., Wong, L., Feng, M., & Tan, Y-P. (2005). Relative risk and odds ratio: A data mining perspective. *Symposium on Principles of Database Systems* (pp. 368-377).

Manku, G. S., & Motwani, Q. (2002). Approximate frequency counts over data streams. *VLDB* (pp. 346-357).

Mannila. H.. & Toivonen, H. (1997). Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, *1*(2), 241-258.

Metwally, A., Agrawal, D., & Abbadi, A. E. (2005) Efficient computation of frequent and top-k elements in data streams. *International Conference on Data Theory* (pp. 398-412).

Pasquier N., Bastide Y., Taouil R., & Lakhal, L. (1999). Efficient mining of association rules using closed itemset lattices. *Information Systems*, *24*(1), 25-46.

Pei ,J., Han, J. & Mao R. (2000). CLOSET: An efficient algorithm for mining frequent closed itemsets. *SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery 2000* (pp 21-30).

Silverstein, C., Brin, S., & Motwani, R. (1998). Scalable techniques for mining causal structures. *Data Mining and Knowledge Discovery*, *4*(2), 163-192.

Veloso, A. A., Meira ,W. Jr.,Carvalho, M. B., Possas, B., Parthasarathy, S., & Zaki, M. J. (2002). Mining frequent itemsets in evolving databases. *SIAM International Conference on Data Mining*, 2002.

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

ENDNOTES

[1] This work is partially supported by an A*STAR SERC PSF grant, a MOE AcRF Tier 1 grant, and an A*STAR AGS scholarship.

[2] The *CanTree* algorithm in our experimental studies is implemented by us based on Leung et al. 2007.