A Survey on Algorithms for Mining Frequent Itemsets over Data Streams

James Cheng Yiping Ke Wilfred Ng
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{csjames, keyiping, wilfred}@cse.ust.hk

Abstract

The increasing prominence of data streams arising in a wide range of advanced applications such as fraud detection and trend learning has led to the study of online mining of frequent itemsets (FIs). Unlike mining static databases, mining data streams poses many new challenges. In addition to the one-scan nature, the unbounded memory requirement and the high data arrival rate of data streams, the combinatorial explosion of itemsets exacerbates the mining task. The high complexity of the FI mining problem hinders the application of the stream mining techniques. We recognize that a critical review of existing techniques is needed in order to design and develop efficient mining algorithms and data structures that are able to match the processing rate of the mining with the high arrival rate of data streams.

Within a unifying set of notations and terminologies, we describe in this paper the efforts and main techniques for mining data streams and present a comprehensive survey of a number of the state-of-the-art algorithms on mining frequent itemsets over data streams. We classify the stream-mining techniques into two categories based on the window model that they adopt in order to provide insights into how and why the techniques are useful. Then, we further analyze the algorithms according to whether they are *exact* or *approximate* and, for approximate approaches, whether they are *false-positive* or *false-negative*. We also discuss various interesting issues, including the merits and limitations in existing research and substantive areas for future research.

1 Introduction

Frequent itemset mining [1] has been well recognized to be fundamental to many important data mining tasks, such as associations [2], correlations [7, 36], sequences [3], episodes [35], classifiers [32] and clusters [43]. There is a great amount of work that studies mining frequent itemsets on static databases and many efficient algorithms [22] have been proposed.

Recently, the increasing prominence of data streams has led to the study of online mining of frequent itemsets, which is an important technique that is essential to a wide range of emerging applications [20], such as web log and click-stream mining, network traffic analysis, trend analysis and fraud detection in telecommunications data, e-business and stock market analysis, and sensor networks. With the rapid emergence of these new application domains, it has become increasingly difficult to conduct advanced analysis and data mining over fast-arriving and large data streams in order to capture interesting trends, patterns and exceptions.

Unlike mining static databases, mining data streams poses many new challenges. First, it is unrealistic to keep the entire stream in the main memory or even in a secondary storage area, since a data stream comes continuously and the amount of data is unbounded. Second, traditional methods of mining on stored datasets by multiple scans are infeasible, since the streaming data is passed only once. Third, mining streams requires fast, real-time processing in order to keep up with the high data arrival rate and mining results are expected to be available within short response times. In addition, the combinatorial explosion[1] of itemsets exacerbates mining frequent itemsets over streams in terms of both memory consumption and processing efficiency. Due to these constraints, research studies have been conducted on approximating mining results, along with some reasonable guarantees on the quality of the approximation.

In this paper, we survey a number of representative state-of-the-art algorithms [34, 9, 10, 31, 21, 11, 45, 17, 29, 18] on mining frequent itemsets, frequent maximal itemsets [24], or frequent closed itemsets [37] over data streams. We organize the algorithms into two categories based on the window model that they adopt: the *landmark window* or

[1]Given a set of items, $\mathcal{I}$, the possible number of itemsets can be up to $2^{|\mathcal{I}|} - 1$.

the *sliding window.* Each window model is then classified as *time-based* or *count-based.* According to the number of transactions that are updated each time, the algorithms are further categorized into *update per transaction* or *update per batch.* Then, we classify the mining algorithms into two categories: *exact* or *approximate.* We also classify the approximate algorithms according to the results they return: the *false-positive* approach or the *false-negative* approach. The false-positive approach [34, 9, 31, 10, 21, 11, 29] returns a set of itemsets that includes all frequent itemsets but also some infrequent itemsets, while the false-negative approach [45] returns a set of itemsets that does not include any infrequent itemsets but misses some frequent itemsets. We discuss the different issues raised from the different window models and the nature of the algorithms. We also explain the underlying principle of the ten algorithms and analyze their merits and limitations.

The rest of this paper is organized as follows. We first give the preliminaries and the notation in Section 2. Then, in Sections 3 and 4, we discuss the underlying techniques of various representative algorithms for mining frequent itemsets and identify their main features such as window models, update modes and approximation types. We present an overall analysis of the algorithms in Sections 5 and discuss some future research and related work in Section 6. Finally, we conclude the paper in Section 7.

2 Preliminaries

Let $\mathcal{I}$ be a set of items. An *itemset* (or a *pattern*), $I = \{x_1, x_2, \ldots, x_k\}$, is a subset of $\mathcal{I}$. An itemset consisting of k items is called a *k-itemset* and is written as $x_1x_2\cdots x_k$. We assume that the items in an itemset are lexicographically ordered. A *transaction* is a tuple, (tid, Y), where tid is the ID of the transaction and Y is an itemset. The transaction *supports* an itemset, X, if $Y \supseteq X$. For simplicity, we may omit the tid when the ID of a transaction is irrelevant to the underlying idea of a mining algorithm.

A *transaction data stream* is a sequence of incoming transactions and an excerpt of the stream is called a *window.* A window, W, can be (1) either *time-based* or *count-based*, and (2) either a *landmark window* or a *sliding window.* W is time-based if W consists of a sequence of fixed-length time units, where a variable number of transactions may arrive within each time unit. W is count-based if W is composed of a sequence of batches, where each batch consists of an equal number of transactions. W is a landmark window if $W = \langle T_1, T_2, \ldots, T_\tau \rangle$; W is a sliding window if $W = \langle T_{\tau-w+1}, \ldots, T_\tau \rangle$, where each T_i is a time unit or a batch, T_1 and T_τ are the *oldest* and the *current* time unit or batch, and w is the number of time units or batches in the sliding window, depending on whether W is time-based or count-based. Note that a count-based window can also be captured by a time-based window by assuming that a uniform number of transactions arrive within each time unit.

The *frequency* of an itemset, X, in W, denoted as $\mathit{freq}(X)$, is the number of transactions in W that support X. The *support* of X in W, denoted as $\mathit{sup}(X)$, is defined as $\mathit{freq}(X)/N$, where N is the total number of transactions received in W. X is a *Frequent Itemset* (*FI*) in W, if $\mathit{sup}(X) \geq \sigma$, where σ $(0 \leq \sigma \leq 1)$ is a user-specified *minimum support threshold*. X is a *Frequent Maximal Itemset* (*FMI*) in W, if X is an FI in W and there exists no itemset Y in W such that $X \subset Y$. X is a *Frequent Closed Itemset* (*FCI*) in W, if X is an FI in W and there exists no itemset Y in W such that $X \subset Y$ and $\mathit{freq}(X) = \mathit{freq}(Y)$.

Given a transaction data stream and a minimum support threshold, σ, the problem of *FI/FMI/FCI mining over a window, W, in the transaction data stream* is *to find the set of all FIs/FMIs/FCIs over W*.

To mine FIs/FMIs/FCIs over a data stream, it is necessary to keep not only the FIs/FMIs/FCIs, but also the infrequent itemsets, since an infrequent itemset may become frequent later in the stream. Therefore, existing approximate mining algorithms [34, 9, 10, 11, 21, 31, 45] use a *relaxed* minimum support threshold (also called a *user-specified error parameter*), ϵ, where $0 \leq \epsilon \leq \sigma \leq 1$, to obtain an extra set of itemsets that are potential to become frequent later. We call an itemset that has support no less than ϵ a *sub-frequent itemset* (*sub-FI*).

Example 2.1 Table 1 records the transactions that arrive on the stream within three successive time units or batches, each of which consists of five transactions.

Let $\sigma = 0.6$ and $\epsilon = 0.4$. Assume that the model is landmark window. We obtain three consecutive windows, $W_1 = \langle T_1 \rangle$, $W_2 = \langle T_1, T_2 \rangle$ and $W_3 = \langle T_1, T_2, T_3 \rangle$. The

T_1	T_2	T_3
abc	bcd	ac
abcd	abc	abc
abx	bc	abcd
abu	abcv	abc
bcy	abd	acwz

Table 1: Transactions (*tid* omitted for brevity) in a Data Stream

minimum frequencies of FI (sub-FI) in W_1, W_2 and W_3 are $5\sigma = 3$ ($5\epsilon = 2$), $10\sigma = 6$ ($10\epsilon = 4$) and $15\sigma = 9$ ($15\epsilon = 6$), respectively. Assume that the model is a sliding window and the window size is $w = 2$. There are two successive windows, $W_1 = \langle T_1, T_2 \rangle$ and $W_2 = \langle T_2, T_3 \rangle$. The minimum frequencies of FI (sub-FI) in both W_1 and W_2 are $10\sigma = 6$ ($10\epsilon = 4$).

In Table 2, we show all the FIs in all the above-described windows, while we also show the sub-FIs that are not FIs in italics. The number in the brackets shows the frequency of the itemset. All the FIs, except `a` and `c` in $\langle T_1 \rangle$ and $\langle T_1, T_2 \rangle$, are also FCIs since they have no superset that has the same frequency in the same window. All the frequent 2-itemsets are also MFIs. We also note that `ac` only becomes an FI in $\langle T_1, T_2, T_3 \rangle$ and $\langle T_2, T_3 \rangle$; therefore, if we do not keep `ac` in $\langle T_1 \rangle$ and $\langle T_1, T_2 \rangle$ in which `ac` is not an FI, we will miss `ac` in $\langle T_1, T_2, T_3 \rangle$ and $\langle T_2, T_3 \rangle$. Thus, it is necessary to use a relaxed minimum support threshold, ϵ, to keep an extra set of sub-FIs that have the potential to become FIs later in the stream.

$\square$

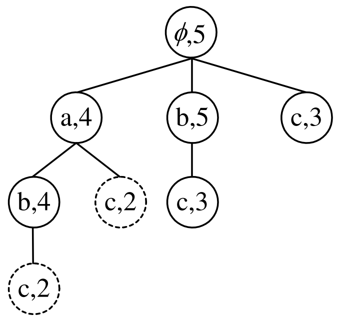


Figure 1: A Prefix Tree Representation of the Sub-FIs over $\langle T_1 \rangle$

$\langle T_1 \rangle$	$\langle T_1, T_2 \rangle$	$\langle T_1, T_2, T_3 \rangle$	$\langle T_2, T_3 \rangle$
`a` (4)	`a` (7)	`a` (12)	`a` (8)
`b` (5)	`b` (10)	`b` (13)	`b` (8)
`c` (3)	`c` (7)	`c` (12)	`c` (9)
`ab` (4)	`ab` (7)	`ab` (10)	`ab` (6)
`bc` (3)	`bc` (7)	`ac` (9)	`ac` (7)
		`bc` (10)	`bc` (7)
ac (2)	*ac* (4)	*abc* (7)	*abc* (5)
abc (2)	*abc* (4)		

Table 2: FIs and sub-FIs

A prevalently used data structure to represent the itemsets is the prefix tree structure [34, 9, 10, 21, 11, 17]. In Figure 1, we show a prefix tree that stores all sub-FIs over $\langle T_1 \rangle$ in Table 2 of Example 2.1. The nodes in the prefix tree are ordered according to the lexicographic order of the items stored in the node. Each path in the prefix tree represents an itemset, where the integer kept in the last node on the path is the frequency of the itemset. For example, the left-most path from the node labelled "`a,4`" to the node labelled "`c,2`" represents the itemset `abc` and the number "2" in the node labelled "`c,2`" is the frequency of `abc`. In Figure 1, the nodes represented by solid circles are FIs, while those represented by dotted circles are sub-FIs that are not FIs.

As we mentioned earlier, most existing studies focus on mining approximate results due to the high complexity of mining data streams. In the approximate mining algorithms we are going to discuss, the frequency (or support) of an itemset, X, is an approximation of its *actual frequency* (or *actual support*). We call this frequency (or support) of X the *computed frequency* (or *computed support*) of X. We denote the computed frequency (or computed support) of X as $\widetilde{freq}(X)$ (or $\widetilde{sup}(X)$) to distinguish it from its actual frequency (or actual support), i.e., $freq(X)$ (or $sup(X)$). Obviously, $\widetilde{freq}(X) \leq freq(X)$ and $\widetilde{sup}(X) \leq sup(X)$. We denote the upper bound for the error in the computed frequency of X as $err(X)$, such that $(freq(X) - err(X)) \leq \widetilde{freq}(X) \leq freq(X)$. In the rest of the paper, when we refer to the frequency (or support) of an itemset, we mean the actual frequency (or actual support) of the itemset.

3 Mining over a Landmark Window

In this section, we describe six algorithms on mining FIs/FMIs over a landmark window. We first discuss three algorithms [34, 31, 45], presented in Sections 3.1 to 3.3, that do not distinguish recent itemsets from old ones and then three others [9, 21, 29], presented in Sections 3.4 and 3.6, that place more importance on recent itemsets of a data stream than the old ones.

3.1 Lossy Counting Algorithm

Manku and Motwani [34] propose the *Lossy Counting* algorithm for computing an approximate set of FIs over the entire history of a stream. The stream is divided into a sequence of *buckets* and each bucket consists of $B = \lceil 1/\epsilon \rceil$ transactions. Each bucket is also given an ID and the ID of the *current* bucket, bid_τ, is assigned as $bid_\tau = \lceil N/B \rceil$, where N is the number of transactions currently in the stream. Lossy Counting processes a batch of transactions arriving on the stream at a time, where each batch contains β buckets of transactions.

Lossy Counting maintains a set of tuples, $\mathcal{D}$, for the stream of transactions. Each tuple in $\mathcal{D}$ is of the form $(X, \widetilde{freq}(X), err(X))$, where X is a sub-FI, $\widetilde{freq}(X)$ is assigned as the frequency of X, since X is inserted into $\mathcal{D}$ and $err(X)$ is assigned as an upper bound of the frequency of X before X is inserted into $\mathcal{D}$. Lossy Counting updates $\mathcal{D}$ according to the following two categories:

- *UpdateEntry*: For each tuple, $(X, \widetilde{freq}(X), err(X)) \in \mathcal{D}$, add the frequency of X in the current batch to $\widetilde{freq}(X)$. If $(\widetilde{freq}(X) + err(X)) < bid_\tau$, we delete this tuple.

- *AddEntry*: If the frequency of an itemset, X, is at least β in the current batch and X is not in any tuple in $\mathcal{D}$, add a new tuple to $\mathcal{D}$, where $\widetilde{freq}(X)$ is assigned as the frequency of X in the current batch and $err(X) = (bid_\tau - \beta)$.

Since $B = \lceil 1/\epsilon \rceil$ and $bid_\tau = \lceil N/B \rceil$, we have $bid_\tau \approx \epsilon N$. An itemset, X, is removed from $\mathcal{D}$ if $(\widetilde{freq}(X) + err(X)) < bid_\tau$. Since $freq(X) \leq (\widetilde{freq}(X) + err(X))$

and $bid_\tau = \epsilon N$, we have $freq(X) < \epsilon N$ if an itemset, X, is not in any tuple in $\mathcal{D}$. At the point just before the current batch, we have $\epsilon N = (bid_\tau - \beta)$, since there are β buckets in the current batch. Therefore, when a new itemset, X, is added to $\mathcal{D}$, its frequency before the current batch can be at most $\epsilon N = (bid_\tau - \beta)$ and thus $err(X) = (bid_\tau - \beta)$. As a result, Lossy Counting ensures that $\forall X$, if $freq(X) \geq \epsilon N$, $(X, \widetilde{freq}(X), err(X)) \in \mathcal{D}$. Finally, Lossy Counting outputs all tuples, $(X, \widetilde{freq}(X), err(X)) \in \mathcal{D}$ if $\widetilde{freq}(X) \geq (\sigma - \epsilon)N$.

Implementation: The implementation of Lossy Counting in [34] consists of the following three main modules: *Buffer*, *Trie* and *SetGen*.

The Buffer module repeatedly fills the available main memory with as many incoming transactions as possible. The module computes the frequency of every item, $x \in \mathcal{I}$, in the current batch and prunes those items whose frequency is less than ϵN. Then, the remaining items in the transactions are sorted.

The Trie module maintains the set, $\mathcal{D}$, as a forest of prefix trees, where the prefix trees are ordered by the labels of their roots. A node, v, in a prefix tree corresponds to an tuple, $(X, \widetilde{freq}(X), err(X)) \in \mathcal{D}$ and v is also assigned a label as the last item in X so that X is represented by the label path from the root to v. Manku and Motwani implement the Trie forest as an array of tuples $(X, \widetilde{freq}(X), err(X), level)$ that correspond to the pre-order traversal of the forest, where the *level* of a node is the distance of the node from the root. The Trie array is maintained as a set of chunks. On updating the Trie array, a new Trie array is created and chunks from the old Trie are freed as soon as they are not required.

The SetGen module generates the itemsets that are supported by the transactions in the current batch. In order to avoid the combinatorial explosion of the itemsets, the module applies an Apriori-like pruning rule [2] such that no superset of an itemset will be generated if the itemset has a frequency less than β in the current batch. SetGen employs a priority queue, called *Heap*, which initially contains pointers to the smallest items of all transactions in the buffer. Pointers pointing to identical items are grouped together as a single entry in Heap. SetGen repeatedly processes the smallest item in Heap to generate a 1-itemset. If this 1-itemset is in Trie after the AddEntry or the UpdateEntry operation is utilized, SetGen is recursively invoked with a new Heap created out of the items that follow the smallest items in the same transactions. During each call of SetGen, qualified old itemsets are copied to the new Trie array according to their orders in the old Trie array, while at the same time new itemsets are added to the new Trie array in lexicographic order. When the recursive call returns, the smallest entry in Heap is removed and the recursive process continues with the next smallest item in Heap.

Merits and Limitations: A distinguishing feature of the Lossy Counting algorithm is that it outputs a set of itemsets that have the following guarantees:

- All FIs are outputted. There are no false-negatives.
- No itemset whose actual frequency is less than $(\sigma - \epsilon)N$ is outputted.
- The computed frequency of an itemset is less than its actual frequency by at most ϵN.

However, using a relaxed minimum support threshold, ϵ, to control the quality of the approximation of the mining result leads to a dilemma. The smaller the value of ϵ, the more accurate is the approximation but the greater is the number of sub-FIs generated, which requires both more memory space and more CPU processing power. However, if ϵ approaches σ, more false-positive answers will be included in the result, since all sub-FIs whose computed frequency is at least $(\sigma - \epsilon)N \approx 0$ are outputted while the computed frequency of the sub-FIs can be less than their actual frequency by as much as σN. We note that this problem also exists in other mining algorithms [9, 10, 11, 21, 31, 29] that use a relaxed minimum support threshold to control the accuracy of the mining result.

3.2 Item-Suffix Frequent Itemset Forest

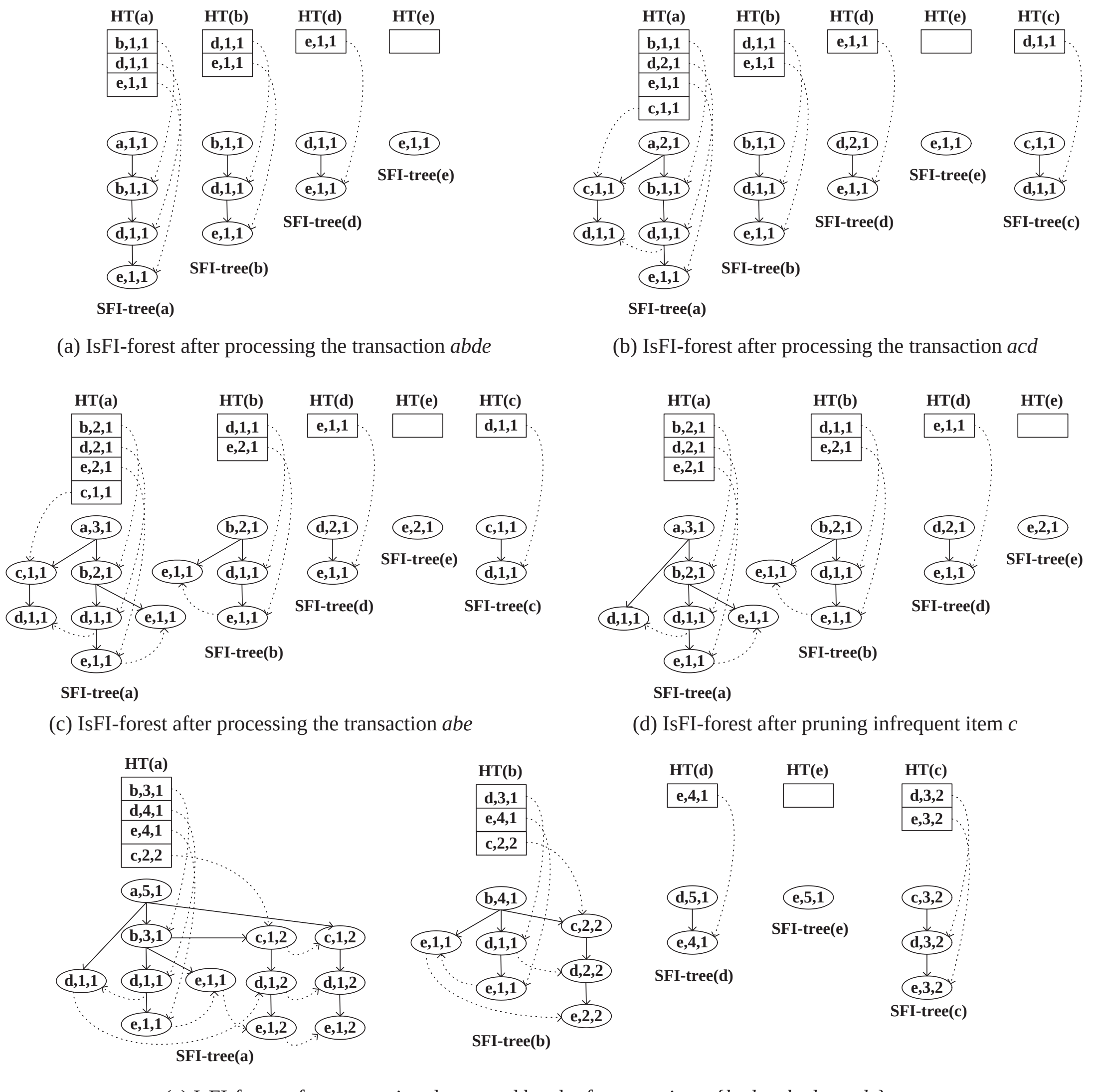
Li et al. [31] develop a prefix-tree-based, in-memory data structure, called *Item-suffix Frequent Itemset forest* (*IsFI-forest*), based on which an algorithm, called *DSM-FI*, is devised to mine an approximate set of FIs over the entire history of a stream.

DSM-FI also uses a relaxed minimum support threshold, ϵ, to control the accuracy of the mining results. All generated sub-FIs are kept in IsFI-forest, which consists of a set of paired components, *Header Table* (*HT*) and *Sub-Frequent Itemset tree* (*SFI-tree*). For each sub-frequent item, x, DSM-FI constructs an HT and an SFI-tree. Then, for each unique item, y, in the set of sub-FIs that are prefixed by x, DSM-FI inserts an entry (y, $\widetilde{freq}(y)$, *batch-id*, *head-link*) into the HT of x or increments $\widetilde{freq}(y)$ if y already exists in the HT, where *batch-id* is the ID of the processing batch into which the entry is inserted and *head-link* points to first node created due to y in the SFI-tree of x. Each node in the SFI-tree has four fields, *item*, $\widetilde{freq}$, *batch-id*, and *node-link*. Note that the edges between the parent and children of the tree are implicitly assumed. A path from the root node of x's SFI-tree to a node, v, represents a sub-FI, whose prefix and suffix are x and $v.item$, respectively. Thus, $v.\widetilde{freq}$ keeps the computed frequency of the sub-FI and $v.batch\text{-}id$ is the ID of the batch on processing which the sub-FI is inserted into the SFI-tree. If there will be another node, whose *item* is the same as $v.item$, inserted into the SFI-tree, then $v.node\text{-}link$ will point to that node; otherwise, $v.node\text{-}link$ is a NULL-pointer. Therefore, by tracing the IsFI-forest, we can find all sub-FIs and their computed frequencies.

For each transaction, $x_1x_2\cdots x_{k-1}x_k$, in an incoming batch, DSM-FI projects the transaction into k *item-suffix transactions* as follows: $x_1x_2\cdots x_{k-1}x_k$, $x_2\cdots x_{k-1}x_k$, $\ldots$, $x_{k-1}x_k$ and x_k. Then, these item-suffix transactions are inserted into the SFI-trees of the corresponding items, $x_1, x_2, \ldots, x_{k-1}, x_k$. If the item-suffix transaction already exists in the SFI-tree, DSM-FI simply increments its computed frequency. Meanwhile, DSM-FI inserts an entry for each item in the item-suffix transaction that does not exist in the HT, while it increments the computed frequency of other items that are in the HT.

Periodically, DSM-FI prunes those itemsets whose computed support is less than ϵ. The pruning may require reconnection of the nodes in an SFI-tree, since each root-to-node path, P, that represents a sub-FI X, also carries the frequency information of a set of sub-FIs that are subsets of X and have the same prefix as X.

Example 3.1 Figure 2 shows an example of constructing an IsFI-forest and pruning infrequent itemsets from the IsFI-forest. We consider constructing an IsFI-forest for two successive batches of transactions, {`abde`, `acd`, `abe`} and {`bcde`, `abcde`, `acde`}, with the IDs being equal to 1 and 2, respectively.



(a) IsFI-forest after processing the transaction *abde*

(b) IsFI-forest after processing the transaction *acd*

(c) IsFI-forest after processing the transaction *abe*

(d) IsFI-forest after pruning infrequent item *c*

(e) IsFI-forest after processing the second batch of transactions, {*bcde, abcde, acde*}

Figure 2: An Example of Constructing an IsFI-forest

The IsFI-forest in each of the subfigures is a set of paired HTs and SFI-trees, for each of the unique items, `a`, `b`, `c`, `d` and `e`. The HT entries are linked to the corresponding nodes in the SFI-tree via the head-links, while nodes of the same item in an SFI-tree are linked via node-links in the order of their insertion time. Both head-links and node-links are represented by dotted lines in the subfigures, while the parent-child edges are represented by solid lines.

Figures 2(a) to (c) show the IsFI-forest constructed after processing the transactions in the first batch, {`abde`, `acd`, `abe`}. For the transaction `abde`, DSM-FI inserts three entries, (`b`,1,1), (`d`,1,1) and (`e`,1,1), to `a`'s HT and links the entries to the corresponding nodes in `a`'s SFI-tree, as shown in Figure 2(a). Similarly for `abde`'s item-suffix transactions, `bde`, `de` and `e`, the HT entries and the SFI-tree nodes are inserted into the HT and the SFI-tree of the items, `b`, `d` and `e`, respectively.

In Figure 2(b), we can see that after adding the transaction `acd`, the frequencies of the corresponding HT entries and SFI-tree nodes are incremented to 2 and a new pair of HT and SFI-tree is added for the new item `c`. Since the path `acd` does not exist in `a`'s SFI-tree, it is added and a node-link from the old node (`d`,1,1) is linked to the new node (`d`,1,1). A similar update is performed for the next transaction, `abe`, as shown in Figure 2(c).

After processing the first batch, DSM-FI performs pruning of the IsFI-forest. We assume that $\epsilon = 0.35$: that is, all items whose computed frequency is less than 2 will be pruned. Thus, the HT and the SFI-tree of item `c` are pruned from the IsFI-forest in Figure 2(c). Then, we search for `c` in the HTs of the other items. Since the path `acd` in `a`'s SFI-tree actually carries the frequency information of the itemsets, `a`, `ac`, `ad` and `acd`, we need to reconnect `a` to `d` after deleting `c`, as shown in Figure 2(d). Note that after the reconnection, the infrequent itemsets `ac` and `acd` are removed.

We then add the HT and the SFI-tree of item `c` back to the IsFI-forest for the second batch of transactions, {`bcde`, `abcde`, `acde`}. Note that the batch-id of the new HT entries and SFI-tree nodes in Figure 2(e) is 2.

□

The batch-id kept in an HT entry and in an SFI-tree node is used to estimate the actual frequency of the corresponding itemset, X, represented by a node, v, where $\widetilde{freq}(X) = v.\widetilde{freq}$, as described by the following equation. We assume that the number of transactions in each batch is the *batch-size.*

$$freq(X) \;\leq\; (\widetilde{freq}(X) + \;\epsilon\; \cdot \;(v.batch\text{-}id \;-\; 1)\; \cdot\; batch\text{-}size) \tag{1}$$

To output the FIs, DSM-FI first follows the HT of each item, x, to generate a *maximal itemset*[2] that contains x and all items in the HT of x. Next, DSM-FI traverses the SFI-tree of x via the node-links to obtain the computed frequency of the maximal itemset. Then, DSM-FI checks whether the maximal itemset is frequent. This checking is done by testing whether the right side of Equation (1) is no less than σN, where X is the maximal itemset and N is the total number of transactions received so far. If the maximal itemset is frequent, DSM-FI generates the FIs that are subsets of the maximal itemset directly based on the maximal itemset (note that, however, we may still need to compute their exact computed frequency from the HTs and SFI-trees of related items). Otherwise, DSM-FI repeats the process on the $(k-1)$-itemsets that are subsets of the maximal itemset and share the prefix x, assuming the maximal itemset has k items. The process stops when $k = 2$, since the frequent 2-itemsets can be found directly by combining each item in the HT with x.

Example 3.2 Assume that $\sigma = 0.5$: that is, all itemsets whose frequency is no less than $6\sigma = 3$ are FIs. We consider computing the set of FIs from the IsFI-forest shown in Figure 2(e).

We start with the item, `a`, and find the maximal itemset, `abcde`. Since the computed frequency of `abcde` is 1, `abcde` is infrequent. We continue with `abcde`'s subsets that are 4-itemsets and find that none of them is frequent. Then, we continue with the 3-itemsets that are `abcde`'s subsets. We find that both `abe` and `ade` have a computed frequency of 3; hence, we generate their subsets, `a`, `b` `d`, `e`, `ab`, `ad`, `ae`, `be`, `de`, `abe` and `ade`. These subsets are FIs and their computed frequency is then obtained from the HTs and the SFI-trees. Since there is no maximal itemset in `a`'s SFI-tree other than `abcde`, we continue with the next item, `b`, and then other items, in a similar way.
□

Merits and Limitations: An SFI-tree is a more compact data structure than is a prefix tree. For example, if the itemset $x_1x_2x_3$ is an FCI, then it is represented by one path, $\langle x_1, x_2, x_3\rangle$, in an SFI-tree but by two paths, $\langle x_1, x_2, x_3\rangle$ and $\langle x_1, x_3\rangle$, in a prefix tree. However, if $x_1x_2x_3$ is not an FCI, then it is also represented by the same two

[2] A maximal itemset is an itemset that is not a subset of any other itemset.

paths. We remark that the number of FCIs approaches that of FIs when the stream becomes large. Moreover, the compactness of the data structure is paid for by the price of a higher computational cost, since more tree traversals are needed to gather the frequency information of the itemsets.

3.3 A Chernoff-Bound-Based Approach

Yu et al. [45] propose *FDPM*, which is derived from the *Chernoff bound* [16], to approximate a set of FIs over a landmark window. Suppose that there is a sequence of N observations and consider the first n ($n \ll N$) observations as independent coin flips such that $Pr(head) = p$ and $Pr(tail) = 1 - p$. Let r be the number of heads. Then, the expectation of r is np. The Chernoff bound states, for any $\gamma > 0$, are

$$Pr(|r - np| \geq np\gamma) \leq 2e^{\frac{-np\gamma^2}{2}}. \tag{2}$$

By substituting $\bar{r} = r/n$ and $\epsilon = p\gamma$, Equation (2) becomes

$$Pr(|\bar{r} - p| \geq \epsilon) \leq 2e^{\frac{-n\epsilon^2}{2p}}. \tag{3}$$

Let $\delta = 2e^{\frac{-n\epsilon^2}{2p}}$. We obtain

$$\epsilon = \sqrt{\frac{2p\ln(2/\delta)}{n}}. \tag{4}$$

Then, from Equation (3), we derive

$$Pr\{p - \epsilon \leq \bar{r} \leq p + \epsilon\} \geq (1 - \delta). \tag{5}$$

Equation (5) states that the probability of being a "*head*" in n coin flips, $\bar{r}$, is within the range of $[p - \epsilon, p + \epsilon]$ with a probability of at least $(1 - \delta)$. Yu et al. apply the Chernoff bound in the context of mining frequent itemsets as follows. Consider the first n transactions of a stream of N transactions as a sequence of coin flips such that for an itemset, X, $Pr(\textit{a transaction supports } X) = p$. By replacing $\bar{r}$ with $\textit{freq}(X)/n$ (i.e., the support of X in the n transactions) and p with $\textit{freq}(X)/N$ (i.e., the probability of a transaction supporting X in the stream), the following statement is valid: the support of an itemset, X, in the first n transactions of a stream is within $\pm\epsilon$ of the support of X in the entire stream with a probability of at least $(1-\delta)$.

The Chernoff bound is then applied to derive the FI mining algorithm FDPM by substituting p in Equations (4) and (5) with σ. Note that p corresponds to the actual support of an itemset in the stream, which varies for different itemsets, but σ is specified by the user. The rationale for using a constant σ in place of a varying p is based on the heuristic that σ is the minimum support of all FIs, such that if this minimum support is satisfied, then all other higher support values are also satisfied.

The underlying idea of the FDPM algorithm is explained as follows. First, a *memory bound*, $n_0 \approx (2 + 2\ln(2/\delta))/\sigma$, is derived from Equations (4) and (5). Given a probability parameter, δ, and an integer, k. The *batch size*, B, is given as $k \cdot n_0$. Then, for each *batch* of B transactions, FDPM employs an existing non-streaming FI mining algorithm to compute all itemsets whose support in the current batch is no less than $(\sigma - \epsilon_B)$, where $\epsilon_B = \sqrt{(2\sigma \ln(2/\delta))/B}$ (by Equation (4)). The set of itemsets computed are then merged with the set of itemsets obtained so far for the stream. If the total number of itemsets kept for the stream is larger than $c \cdot n_0$, where c is an empirically determined float number, then all itemsets whose support is less than $(\sigma - \epsilon_N)$ are pruned, where N is the number of transactions received so far in the stream and $\epsilon_N = \sqrt{(2\sigma \ln(2/\delta))/N}$ (by Equation (4)). Finally, FDPM outputs those itemsets whose frequency is no less than σN.

Merits and Limitations: Given σ and δ, the Chernoff bound provides the following guarantees on the set of mined FIs:

- All itemsets whose actual frequency is no less than σN are outputted with a probability of at least $(1-\delta)$.
- No itemset whose actual frequency is less than σN is outputted. There are no false-positives.
- The probability that the computed frequency of an itemset equals its actual frequency is at least $(1-\delta)$.

However, Equations (2) to (5) are computed with respect to n of N observations and the result of these n observations is exact, as in the application of the Chernoff bound in sampling [41, 48]. On the contrary, in FDPM, infrequent itemsets are pruned for each new batch of transactions and some sub-FIs are pruned whenever the total number of sub-FIs kept exceeds $c \cdot n_0$. Thus, the result of these n observations (of a stream of N observations) in FDPM is not exact but already approximate, while the error in the approximation due to each pruning will propagate to each of the following approximations. Consequently, the quality of the approximation is degraded and worsen for each propagation of error in the approximations.

3.4 Mining Recent Frequent Itemsets

Chang and Lee [9] propose to use a *decay rate*, d $(0 < d < 1)$, to diminish the effect of old transactions on the mining result. As a new transaction comes in, the frequency of an old itemset is discounted by a factor of d. Thus, the set of FIs returned is called *recent FIs*.

Assume that the stream has received τ transactions, $\langle Y_1, Y_2, \ldots, Y_\tau \rangle$. The *decayed total number of transactions*, N_τ, and the *decayed frequency* of an itemset, $\mathit{freq}_\tau(X)$, are defined as follows:

$$N_\tau = d^{\tau-1} + d^{\tau-2} + \cdots + d^1 + 1 = \frac{1 - d^\tau}{1 - d}. \quad (6)$$

$$\mathit{freq}_\tau(X) = d^{\tau-1} \times w_1(X) + d^{\tau-2} \times w_2(X) + \cdots + d^1 \times w_{\tau-1}(X) + 1 \times w_\tau(X), \quad (7)$$

$$\text{where} \quad w_i(X) = \begin{cases} 1 & \text{if } X \subseteq Y_i, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, whenever a transaction, say the τ-th transaction, Y_τ, arrives in the stream, $N_{\tau-1}$ and $\mathit{freq}_{\tau-1}(X)$ of each existing itemset, X, are discounted by a factor of d. Then, we add one (for the new transaction, Y_τ) to $d \cdot N_{\tau-1}$, which then gives N_τ. If Y_τ supports X, we also add one to $d \cdot \mathit{freq}_{\tau-1}(X)$, which then gives $\mathit{freq}_\tau(X)$; otherwise, $\mathit{freq}_\tau(X)$ is equal to $d \cdot \mathit{freq}_{\tau-1}(X)$. Therefore, the more frequent X is supported by recent transactions, the greater is its decayed frequency $\mathit{freq}_\tau(X)$.

The algorithm, called *estDec*, for maintaining recent FIs is an approximate algorithm that adopts the mechanism in Hidber [26] to estimate the frequency of the itemsets. The itemsets generated from the stream of transactions are maintained in a prefix tree structure, $\mathcal{D}$. An itemset in $\mathcal{D}$ has three fields: $\widetilde{freq}_\tau(X)$, $err_\tau(X)$ and $tid(X)$, where $\widetilde{freq}_\tau(X)$ is X's *computed decayed frequency*, $err_\tau(X)$ is X's *upper bound decayed frequency error*, and $tid(X)$ is the ID of the most recent transaction that supports X.

Except the 1-itemsets, whose computed decayed frequency is the same as their actual decayed frequency, estDec estimates the computed decayed frequency of a new itemset, X ($|X| \geq 2$), from those of X's ($|X|-1$)-subsets, as described by the following equation:

$$\widetilde{freq}_\tau(X) \;\leq\; min(\{\widetilde{freq}_\tau(X') \mid \forall X' \subset X \text{ and } |X'| = |X|-1\}). \tag{8}$$

When X is inserted into $\mathcal{D}$ at the τ-th transaction, estDec requires all its ($|X|-1$)-subsets to be present in $\mathcal{D}$ before the arrival of the τ-th transaction. Therefore, at least ($|X|-1$) transactions are required to insert X into $\mathcal{D}$. When these ($|X|-1$) transactions are the most recent ($|X|-1$) transactions and support X, $\widetilde{freq}_\tau(X')$ in Equation (8) is maximized as is $\widetilde{freq}_\tau(X)$. Thus, we obtain another upper bound for $\widetilde{freq}_\tau(X)$ as follows:

$$\widetilde{freq}_\tau(X) \;\leq\; \epsilon_{ins} \cdot (N_{\tau-|X|+1} \cdot d^{|X|-1}) + \frac{1-d^{|X|-1}}{1-d}. \tag{9}$$

In Equation (9), ϵ_{ins} ($0 \leq \epsilon_{ins} < \sigma$) is the user-specified *minimum insertion threshold* by which estDec inserts an itemset into $\mathcal{D}$ if its decayed support is no less than ϵ_{ins}. The term $\epsilon_{ins} \cdot (N_{\tau-|X|+1} \cdot d^{|X|-1})$ is thus the upper bound for the decayed frequency of X before the recent ($|X|-1$) transactions, while $(1-d^{|X|-1})/(1-d)$ is the decayed frequency of X over the ($|X|-1$) transactions. By combining Equations (8) and (9), estDec assigns the decayed frequency of a new itemset, X, as follows:

$$\widetilde{freq}_\tau(X) = min\Big(min(\{\widetilde{freq}_\tau(X') \mid \forall X' \subset X \text{ and } |X'| = |X|-1\}),$$

$$\epsilon_{ins} \cdot (N_{\tau-|X|+1} \cdot d^{|X|-1}) + \frac{1-d^{|X|-1}}{1-d}\Big). \tag{10}$$

Then, estDec assigns $err_\tau(X) = (\widetilde{freq}_\tau(X) - freq_\tau^{min}(X))$, where $freq_\tau^{min}(X)$ is X's *lower bound decayed frequency* as described by the following equation.

$$freq_\tau^{min}(X) = max(\{f_\tau^{min}(X_i \cup X_j) \mid \forall X_i, X_j \subset X \text{ and } |X_i| = |X_j| = |X| - 1 \text{ and } i \neq j\}), \tag{11}$$

where

$$f_\tau^{min}(X_i \cup X_j) = \begin{cases} max(0, \widetilde{freq}_\tau(X_i) + \widetilde{freq}_\tau(X_j) - \widetilde{freq}_\tau(X_i \cap X_j)) & \text{if } X_i \cap X_j \neq \emptyset; \\ max(0, \widetilde{freq}_\tau(X_i) + \widetilde{freq}_\tau(X_j) - N_\tau) & \text{otherwise.} \end{cases}$$

For each incoming transaction, say the τ-th transaction, Y_τ, estDec first updates N_τ. Then, for each subset, X, of Y_τ that exists in $\mathcal{D}$, estDec computes $\widetilde{freq}_\tau(X) = \widetilde{freq}_{tid(X)}(X) \cdot d^{\tau - tid(X)} + 1$, $err_\tau(X) = err_{tid(X)}(X) \cdot d^{\tau - tid(X)}$, and $tid(X) = \tau$. If $\widetilde{freq}_\tau(X) < \epsilon_{prn} N_\tau$, where ϵ_{prn} $(0 \leq \epsilon_{prn} < \sigma)$ is the user-specified *minimum pruning threshold*, then X and all its supersets are removed from $\mathcal{D}$. However, X is still kept if it is a 1-itemset, since the frequency of a 1-itemset cannot be estimated later if it is pruned, as indicated by Equation (8).

After updating the existing itemsets, estDec inserts new sub-FIs into $\mathcal{D}$. First, each new 1-itemset, X, is inserted into $\mathcal{D}$ with $\widetilde{freq}_\tau(X) = 1$, $err_\tau(X) = 0$, and $tid(X) = \tau$. Then, for each n-itemset, X $(n \geq 2)$, that is a subset of Y_τ and not in $\mathcal{D}$, if all X's $(n-1)$-subsets are in $\mathcal{D}$ before the arrival of Y_τ, then estDec estimates $\widetilde{freq}_\tau(X)$ as described by Equation (10). If $\widetilde{freq}_\tau(X) \geq \epsilon_{ins} N_\tau$, estDec inserts X into $\mathcal{D}$, with $err_\tau(X)$ computed as described previously and $tid(X) = \tau$. Finally, estDec outputs all recent FIs in $\mathcal{D}$ whose decayed frequency is at least σN_τ.

As will be shown in our overall analysis in Section 5, we derive the support error bound of a result itemset X and the false results returned by estDec as follows:

$$\text{Support error bound} = err_\tau(X)/N_\tau. \tag{12}$$

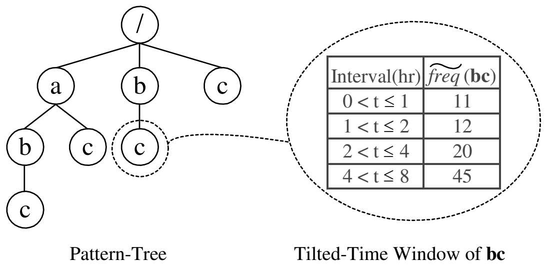
$$\text{False results} = \{X \mid freq_\tau(X) < \sigma N_\tau \leq \widetilde{freq}_\tau(X)\}. \tag{13}$$

Merits and Limitations: The use of a decay rate diminishes the effect of the old and obsolete information of a data stream on the mining result. However, estimating the frequency of an itemset from the frequency of its subsets can produce a large error and the error may propagate all the way from the 2-subsets to the n-supersets, while the upper bound in Equation (9) is too loose. Thus, it is difficult to formulate an error bound on the computed frequency of the resulting itemsets and a large number of false-positive results will be returned, since the computed frequency of an itemset may be much larger than its actual frequency. Moreover, the update for each incoming transaction (instead of a batch) may not be able to handle high-speed streams.

3.5 Mining FIs at Multiple Time Granularities

Giannella et al. [21] propose to mine an approximate set of FIs using a *tilted-time window model* [13]: that is, the frequency of an itemset is kept at a finer granularity for more recent time frames and at a coarser granularity for older time frames. For example, we may keep the frequency of an FI in the last hour, the last two hours, the last four hours, and so on.

The itemsets are maintained in a data structure called *FP-stream*, which consists of two components: the *pattern-tree* and the *tilted-time window*. The pattern-tree is a prefix-tree-based structure. An itemset is represented by a root-to-node path and the end node of the path keeps a tilted-time window that maintains the frequency of the itemset at different time granularities. The pattern-tree can be constructed using the *FP-tree* construction algorithm in [25]. Figure 3 shows an example of an FP-stream. The tilted-time window of the itemset, `bc`, shows that `bc` has a computed frequency of 11, $(11 + 12) = 23$, $(23 + 20) = 43$ and $(43 + 45) = 88$ over the last 1, 2, 4 and 8 hours, respectively. Based on this schema, we need only $(\lceil \log_2(365 \times 24) \rceil + 1) = 15$ frequency records instead of $(365 \times 24) = 8760$ records to keep one year's frequency information. To retrieve the computed frequency of an itemset over the last T time units, the logarithmic tilted-time window guarantees that the time granularity error is at most $\lceil T/2 \rceil$. Updating the frequency records is done by shifting the recent records to merge with older records and the logarithmic scale allows the update to be processed with $\mathcal{O}(1)$ amortized number of shifting and merging operations.



Interval(hr)	$\widetilde{freq}$ (**bc**)
$0 < t \leq 1$	11
$1 < t \leq 2$	12
$2 < t \leq 4$	20
$4 < t \leq 8$	45

Figure 3: An Example of an FP-Stream

To reduce the number of frequency records in the tilted-time windows, the old frequency records of an itemset, X, are pruned as follows. Let $\widetilde{freq}_j(X)$ be the computed frequency of X over a time unit T_j and N_j be the number of transactions received within T_j, where $1 \leq j \leq \tau$. For some m, where $1 \leq m \leq \tau$, the frequency records $\widetilde{freq}_1(X), \ldots, \widetilde{freq}_m(X)$ are pruned if the following condition holds:

$$\begin{aligned} &\exists n \leq \tau, \forall i, 1 \leq i \leq n, \ \ \widetilde{freq}_i(X) \ < \ \sigma N_i \ \ \text{and} \\ &\forall l, 1 \leq l \leq m \leq n, \ \ \sum_{j=1}^{l} \widetilde{freq}_j(X) \ < \ \epsilon \sum_{j=1}^{l} N_j. \end{aligned} \tag{14}$$

The first line in Equation (14) finds a point, n, in the stream such that before that point, the computed frequency of the itemset, X, is less than the minimum frequency required within every time unit, i.e., from T_1 to T_n. Then, the second line in Equation (14) computes the time unit, T_m, within T_1 and T_n, such that at any time unit T_l within T_1 and T_m, the sum of the computed support of X from T_1 to T_l is always less than the relaxed minimum support threshold, ϵ. Thus, the frequency records of X within T_1 and T_m can be considered as unpromising and are pruned.

The pruning ensures that if we return all itemsets whose computed frequency is no less than $(\sigma - \epsilon)N$ over a time interval, T, where N is the total number of transactions received within T, then we will not miss any FIs over T and the computed frequency of the returned itemsets is less than their actual frequency by at most ϵN.

The *FP-streaming* mining algorithm computes a set of sub-FIs at the relaxed minimum support threshold, ϵ, over each batch of incoming transactions by using the FI mining algorithm, *FP-growth* [25]. For each sub-FI X obtained, FP-streaming inserts X into the FP-stream if X is not in the FP-stream. If X is already in the FP-stream, then the computed frequency of X over the current batch is added to its tilted-time window. Next, pruning is performed on the tilted-time window of X and if the window becomes empty, FP-growth stops mining supersets of X by the *Apriori* property [2]. After all sub-FIs mined by FP-growth are updated in the FP-stream, the FP-streaming scans the FP-stream and, for each itemset X visited, if X is not updated by the current batch of transactions, the most recent frequency in X's tilted-time window is recorded as 0. Pruning is then performed on X. If the tilted-time window of some itemset visited is empty (as a result of pruning), the itemset is also pruned from the FP-stream.

Merits and Limitations: The tilted-time window model allows us to answer more expressive time-sensitive queries, at the expense of more than one frequency record kept for each itemset. The tilted-time window also places greater importance on recent data than on old data as does the sliding window model; however, it does not lose the information in the historical data completely. A drawback of the approach is that the FP-stream can become very large over time and updating and scanning such a large structure may degrade the mining throughput.

3.6 Mining Frequent Maximal Itemsets

Lee and Lee [29] extend the estDec algorithm [9] to approximate a set of FMIs. The same decay mechanism applied in estDec is also applied to their work. However, instead of using the simple prefix tree structure, the authors propose a compressed prefix tree structure called *CP-tree*. The structure of the CP-tree is described as follows.

Let $\mathcal{D}$ be the prefix tree used in estDec. Given a *merging gap threshold* δ, where $0 \leq \delta \leq 1$, if all the itemsets stored in a subtree S of $\mathcal{D}$ satisfy the following equation, then S is compressed into a node in the CP-tree.

$$\frac{\widetilde{freq}_\tau(X) - \widetilde{freq}_\tau(Y)}{N_\tau} \leq \delta \ , \tag{15}$$

where X is the root of S and Y is an itemset in S.

Assume S is compressed into a node v in the CP-tree. The node v consists of the following four fields: *item-list*, *parent-list*, $freq_{\tau}^{max}$ and $freq_{\tau}^{min}$, where v.*item-list* is a list of items which are the labels of the nodes in S, v.*parent-list* is a list of locations (in the CP-tree) of the parents of each node in S, $v.freq_{\tau}^{max}$ is the frequency of the root of S and $freq_{\tau}^{min}$ is the frequency of the right-most leaf of S.

Example 3.3 Figure 4 shows a prefix tree and its corresponding CP-tree. The subtree S in the prefix tree is compressed a single node v_3 in the CP-tree as indicated by the dotted arrow. The *item-list* of v_3, $\langle \texttt{b}, \texttt{c} \rangle$, corresponds to the labels, `b` and `c`, of the two nodes in S. The *parent-list*, $\langle v_2.1, v_3.1 \rangle$, means that the parent of the node `b` in S (i.e., `a`) is now in the first position of the *item-list* of v_2 in the CP-tree and the parent of the node `c` in S (i.e., `b`) is now in the first position of the *item-list* of v_3 in the CP-tree. $\square$

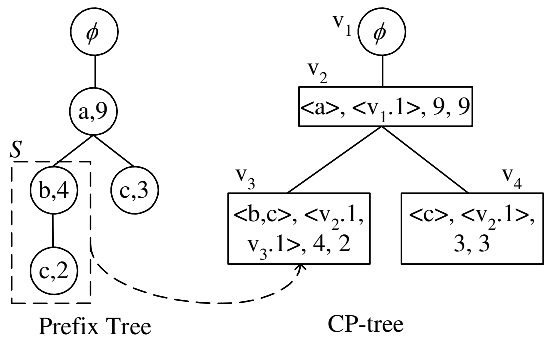


Figure 4: A Prefix Tree and Its Corresponding CP-Tree

The extended mining algorithm is the same as estDec, except that it is now performed on the compressed nodes. Since the nodes are compressed nodes, the FIs are processed using the *item-list* and the *parent-list*. To obtain the set of FMIs, the CP-tree is traversed to output the FMIs (cf. the details of checking of the checking of whether an itemset is an FMI can be consulted from [24]). The frequency of the FMIs is estimated using the same frequency estimation mechanism used in estDec. Note that for mining FMIs, only $freq_{\tau}^{min}$ is needed, while $freq_{\tau}^{max}$ is used to estimate the frequency of other FIs.

Merits and Limitations: The merits and limitations of estDec [9] also apply to [29]. However, the use of the CP-tree results in the reduction of memory consumption, which is vital in mining data streams. The CP-tree can also be used to mine the FIs, however, the error rate of the computed frequency of the FIs, which is estimated from freq_τ^{min} and freq_τ^{max}, will be further increased. Thus, the CP-tree is more suitable for mining FMIs.

4 Mining over a Sliding Window

In this section, we describe three algorithms on mining FIs/FCIs over a sliding window, the first two from Chang and Lee [10, 11] are presented in Section 4.1. They produce approximate results. The last one [17] is presented in Sections 4.2 which computes exact mining results.

4.1 Mining FIs over a Sliding Window

Algorithm *estWin*. Chang and Lee propose the *estWin* algorithm [10] to maintain FIs over a sliding window. The itemsets generated by estWin are maintained in a prefix tree structure (cf. see the discussion of the merits and limitations of [9] in Section 3.4), $\mathcal{D}$. An itemset, X, in $\mathcal{D}$ has the following three fields: $\widetilde{\mathit{freq}}(X)$, $\mathit{err}(X)$ and $\mathit{tid}(X)$, where $\widetilde{\mathit{freq}}(X)$ is assigned as the frequency of X in the current window since X is inserted into $\mathcal{D}$, $\mathit{err}(X)$ is assigned as an upper bound for the frequency of X in the current window before X is inserted into $\mathcal{D}$, and $\mathit{tid}(X)$ is the ID of the transaction being processed, for X is inserted into $\mathcal{D}$.

For each incoming transaction Y with an ID tid_τ, estWin increments the computed frequency of each subset of Y in $\mathcal{D}$. Let N be the number of transactions in the window and tid_1 be the ID of the first transaction in the current window. We prune an itemset X and all X's supersets if (1) $\mathit{tid}(X) \le \mathit{tid}_1$ and $\widetilde{\mathit{freq}}(X) < \lceil \epsilon N \rceil$, or (2) $\mathit{tid}(X) > \mathit{tid}_1$ and $\widetilde{\mathit{freq}}(X) < \lceil \epsilon(N - (\mathit{tid}(X) - \mathit{tid}_1)) \rceil$. Here, the expression $\mathit{tid}(X) \le \mathit{tid}_1$ indicates that X is inserted into $\mathcal{D}$ at some transaction that arrived before the current sliding window, since tid_1 is the ID of the first transaction in the current window, and thus $\widetilde{freq}(X)$ is the actual frequency of X within the current window. On the other hand, the expression $tid(X) > tid_1$ means that X is inserted into $\mathcal{D}$ at some transaction that arrived within the current sliding window and hence the expression $(N-(tid(X)-tid_1))$ returns the number of transactions that arrived within the current window since the arrival of the transaction having the ID $tid(X)$. We note that X itself is not pruned if it is a 1-itemset, since estWin estimates the maximum frequency error of an itemset based on the computed frequency of its subsets [26] and thus the frequency of a 1-itemset cannot be estimated again if it is deleted.

After updating and pruning existing itemsets, estWin inserts new itemsets into $\mathcal{D}$. It first inserts all new 1-itemsets, X, into $\mathcal{D}$, with $\widetilde{freq}(X) = 1$, $err(X) = 0$ and $tid(X) = tid_\tau$. Then, for each new itemset, $X \subseteq Y$ ($|X| \geq 2$), if all X's subsets having size $(|X|-1)$ (or simply $(|X|-1)$-subsets) are in $\mathcal{D}$ before the arrival of Y, then estWin inserts X into $\mathcal{D}$. estWin assigns $\widetilde{freq}(X) = 1$ and $tid(X) = tid_\tau$ and estimates $err(X)$ as described by the following equation:

$$err(X) = min\Big(min(\{(\widetilde{freq}(X') + err(X')) \mid \forall X' \subset X \text{ and } |X'| = |X| - 1\}) - 1, \lfloor \epsilon(w - |X|) \rfloor + |X| - 1\Big). \quad (16)$$

For each expiring transaction of the sliding window, those itemsets in $\mathcal{D}$ that are subsets of the transaction are traversed. For each itemset, X, being visited, if $tid(X) \leq tid_1$, $\widetilde{freq}(X)$ is decreased by 1; otherwise, no change is made since the itemset is inserted by a transaction that comes later than the expiring transaction. Then, pruning is performed on X as described before.

Finally, for each itemset, X, in $\mathcal{D}$, estWin outputs X as an FI if (1) $tid(X) \leq tid_1$ and $\widetilde{freq}(X) \geq \sigma N$, or (2) $tid(X) > tid_1$ and $(\widetilde{freq}(X) + err(X)) \geq \sigma N$.

For our overall analysis in Section 5, we derive the support error bound of a resulting itemset, X, and the false results returned by estWin as follows:

$$\text{Support error bound} = err(X)/N. \quad (17)$$

$$\text{False results} = \{X \mid \mathit{freq}(X) < \sigma N \leq (\widetilde{\mathit{freq}}(X) + \mathit{err}(X))\}. \tag{18}$$

A Lossy-Counting-Based Algorithm. Chang and Lee also propose another similar method [11] to maintain FIs over a sliding window based on the estimation mechanism of the Lossy Counting algorithm [34]. The same prefix tree structure, $\mathcal{D}$, is used to keep the itemsets. An itemset, X, in $\mathcal{D}$ keeps $\widetilde{\mathit{freq}}(X)$, which is assigned as the frequency of X in the current window since X is inserted into $\mathcal{D}$, and $\mathit{tid}(X)$, which is the ID of the transaction under processing when X is inserted into $\mathcal{D}$.

For each incoming transaction, Y, with an ID tid_τ, the algorithm increments the computed frequency of each subset of Y in $\mathcal{D}$ and inserts every new itemset, $X \subseteq Y$, into $\mathcal{D}$ with $\widetilde{\mathit{freq}}(X) = 1$ and $\mathit{tid}(X) = \mathit{tid}_\tau$.

For each subset, X, of an expiring transaction, where X exists in $\mathcal{D}$, if $\mathit{tid}(X) \leq \mathit{tid}_1$, then $\widetilde{\mathit{freq}}(X)$ is decreased by 1. We prune X and all X's supersets if (1) $\mathit{tid}(X) \leq \mathit{tid}_1$ and $\widetilde{\mathit{freq}}(X) < \lceil \epsilon N \rceil$, or (2) $\mathit{tid}(X) > \mathit{tid}_1$ and $(\widetilde{\mathit{freq}}(X) + \lfloor \epsilon(\mathit{tid}(X) - \mathit{tid}_1) \rfloor) < \lceil \epsilon N \rceil$, where $(\mathit{tid}(X) - \mathit{tid}_1)$ is the maximum possible frequency of X in the current window before the arrival of the transaction having the ID $\mathit{tid}(X)$.

Finally, for each itemset, X, in $\mathcal{D}$, X is outputted as an FI, if (1) $\mathit{tid}(X) \leq \mathit{tid}_1$ and $\widetilde{\mathit{freq}}(X) \geq \sigma N$, or (2) $\mathit{tid}(X) > \mathit{tid}_1$ and $(\widetilde{\mathit{freq}}(X) + \lfloor \epsilon(\mathit{tid}(X) - \mathit{tid}_1) \rfloor) \geq \sigma N$.

Merits and Limitations: The sliding window model captures recent pattern changes and trends. However, performing the update for each incoming and expiring transaction is usually much less efficient than batch-processing, especially in a large search space as in the case of FI mining using a relaxed minimum support threshold. Thus, these methods may not be able to cope with high-speed data streams that involve millions of transactions generated from some real-life applications.

4.2 Mining FCIs over a Sliding Window

Chi et al. [17, 18] propose the *Moment* algorithm to incrementally update the set of FCIs over a sliding window. They design an in-memory prefix-tree-based structure, called the *Closed Enumeration Tree* (*CET*), to maintain a dynamically selected set of itemsets over a sliding-window.

Let v_X be a node representing the itemset X in the CET. The dynamically selected set of itemsets (nodes) are classified into the following four types.

- *Infrequent Gateway Nodes* (*IGN*): v_X is an IGN if (1) X is infrequent, (2) v_Y is the parent of v_X and Y is frequent, and (3) if v_Y has a sibling, $v_{Y'}$, such that $X = Y \cup Y'$, then Y' is frequent.

- *Unpromising Gateway Nodes* (*UGN*): v_X is a UGN if (1) X is frequent, and (2) $\exists Y$ such that Y is an FCI, $Y \supset X$, $\mathit{freq}(Y) = \mathit{freq}(X)$ and Y is before X according to the lexicographical order of the itemsets.

- *Intermediate Nodes* (*IN*): v_X is an IN if (1) X is frequent, (2) v_X is the parent of v_Y such that $\mathit{freq}(Y) = \mathit{freq}(X)$, and (3) v_X is not a UGN.

- *Closed Nodes* (*CN*): v_X is a CN if X is an FCI.

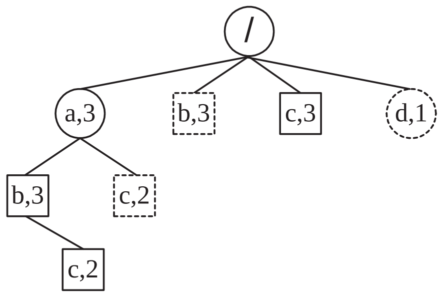


Figure 5: An Example of a CET

Example 4.1 Figure 5 shows an example of a CET constructed for the set of transactions, {`cd`, `ab`, `abc`, `abc`}, where $\sigma = 0.5$ and the number of transactions in each sliding window is 4. The CET is similar to the prefix tree shown in Example 2.1 in Section 2, except that there are four types of nodes. The nodes represented by dotted circles are IGNs, those by dotted squares are UGNs, those by solid circles are INs, and those by solid squares are CNs.

□

By the *Apriori* property [2], all supersets of an infrequent itemset are not frequent. Thus, an IGN, v_X, has no descendants and there is no node, v_Y, in the CET such that $Y \supset X$. If v_X is a UGN, then none of v_X's descendants is a CN; otherwise v_X is a CN but not a UGN. A UGN, v_X, also has no descendants, since no CNs can be found there. Thus, not all itemsets need to be kept in the CET, even though Moment computes the exact mining result.

For each incoming transaction, Moment traverses the parts of the CET that are related to the transaction. For each node, v_X, visited, Moment increments its frequency and performs the following updates to the CET according to the change in v_X's node type:

- v_X is an IGN: If X now becomes frequent, then (1) for each left sibling v_Y of v_X, Moment checks if new children should be created for v_Y as a join result of X and Y; and (2) Moment checks if new descendants of v_X should be created.
- v_X is a UGN: If v_X now becomes an IN or a CN, then Moment checks if new descendants of v_X should be created.
- v_X is an IN: v_X may now become a CN but no other update is made to the CET due to v_X.
- v_X is a CN: v_X will remain a CN and no update is made to the CET due to v_X.

Note that for any node created for a new itemset, Moment needs to scan the current window to compute its frequency.

Example 4.2 Assume that we have an incoming transaction `acd`. Now, we update the CET in Figure 5 to give the CET in Figure 6. The frequency of all subsets of `acd` in Figure 5 is incremented by one. The IN labelled "`a,3`" and the UGN labelled "`c,2`" in Figure 5 now become CNs in Figure 6, since they have no supersets that have the same frequency as theirs. The IGN labelled "`d,1`" now becomes frequent; as a consequence, an IGN "`d,1`" and a CN "`d,2`" are added to the CET. The IGN "`d,1`" in Figure 5 becomes a UGN "`d,2`" in Figure 6 because of the CN "`d,2`", which represents an FCI `cd`. The item `b` is not in `acd` and thus the subtree rooted at `b` in Figure 5 is irrelevant to the update.

□

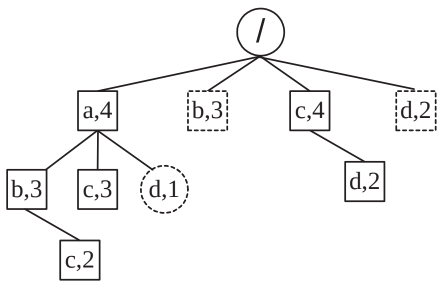


Figure 6: An Updated CET Formed by Adding `acd` to the CET in Figure 5

When a transaction expires from the sliding window, Moment traverses the parts of the CET that are related to the transaction. For each node, v_X, visited, Moment decrements its frequency and performs the following updates to the CET.

- X is infrequent: v_X remains an IGN and no update is made to the CET due to v_X.
- X is frequent:
  - If X now becomes infrequent, then v_X becomes an IGN. Moment first prunes all v_X's descendants. Then, those children of v_X's left-sided siblings that are obtained by joining with v_X are updated recursively.
  - If X remains frequent: if v_X now becomes a UGN, then Moment prunes all v_X's descendants; otherwise, we only need to update the node type of v_X if it changes from a CN to an IN and no other update is made to the CET due to v_X.

Example 4.3 Assume the transaction `cd` has expired. We now update the CET in Figure 6 to give the CET in Figure 7. The frequency of all subsets of `cd` in Figure 6 is decremented by one. The UGN "`d,2`" in Figure 6 now becomes an IGN "`d,1`" in Figure 7. As a consequence, the IGN "`d,1`" and the CN "`d,2`" in Figure 6 are deleted. The CN "`c,4`" in Figure 6 also becomes a UGN "`c,3`" in Figure 7 because it has the same frequency as its superset CN "`c,3`". Other nodes in Figure 6 are irrelevant to the update and thus unchanged in Figure 7.

□

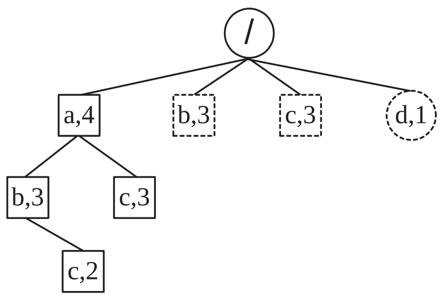


Figure 7: Deleting the Transaction `cd` from the CET in Figure 6

Merits and Limitations: The merit of Moment is that it computes the exact set of FCIs over a sliding window. Although an update to a node may result in a propagation of the node insertion and deletion in the CET, most of the nodes related to an incoming or expiring transaction do not change their type often. Therefore, the average update cost in the CET is small. However, since Moment performs the update for each incoming transaction, it may not be efficient enough to handle bursts and very high-speed streams. Moreover, the CET can be huge for a large window, even though a subset of the itemsets can be derived from IGNs and UGNs.

5 An Overall Analysis

In this section, we first present in Table 3 a summary of the nine algorithms on mining FIs/MFIs/FCIs over data streams [34, 31, 45, 9, 21, 29, 10, 11, 17] that we have discussed in the previous sections of this paper. Then, we present an overall analysis of these algorithms by referencing Table 3.

5.1 Window Model and Update Interval

Among the nine algorithms mentioned in Table 3, six adopt a landmark window model. While the first three landmark window algorithms [34, 31, 45] lose the time information of the itemsets mined, the damped landmark window model [9, 29] favors recent itemsets by diminishing exponentially the effect of old transactions and the tilted landmark window model [21] partitions the window according to a logarithmic scale with the recent frequency of an itemset recorded at a finer granularity. The other three

Representative Work	Window Model	Update Interval	Approximation Type	Support Error Bound	False Results
Manku and Motwani [34]	Landmark Count-Based	Per Batch	False-Positive	ϵ	$\{X \mid (\sigma - \epsilon) \leq sup(X) < \sigma\}$
Li et al. [31]	Landmark Time-Based	Per Batch	False-Positive	ϵ	$\{X \mid (\sigma - \epsilon) \leq sup(X) < \sigma\}$
Yu et al. [45]	Landmark Count-Based	Per Batch	False-Negative	$Pr(0) \geq (1 - \delta)$	$Pr(0) \geq (1 - \delta)$
Chang and Lee [9]	Damped Landmark Count-Based	Per Transaction	False-Positive	Equation (12) Section 3.4	Equation (13) Section 3.4
Giannella et al. [21]	Tilted-Time Landmark Time-Based	Per Batch	False-Positive	ϵ	$\{X \mid (\sigma - \epsilon) \leq sup(X) < \sigma\}$
Lee and Lee [29]	Damped Landmark Count-Based	Per Transaction	False-Positive	Equation (12) Section 3.4	Equation (13) Section 3.4
Chang and Lee [10]	Sliding Count-Based	Per Transaction	False-Positive	Equation (17) Section 4.1	Equation (18) Section 4.1
Chang and Lee [11]	Sliding Count-Based	Per Transaction	False-Positive	ϵ	$\{X \mid (\sigma - \epsilon) \leq sup(X) < \sigma\}$
Chi et al. [17]	Sliding Count-Based	Per Transaction	Exact	0	0

Table 3: An Overview of FI/FCI Mining Algorithms

algorithms [10, 11, 17] consider the most recent excerpt of a stream by adopting a sliding window model.

Out of the nice window models, two [31, 21] are time-based while the other are count-based. In general, time-based windows are more flexible than count-based windows, since they do not need to accumulate a fixed number of transactions in buffer before the mining can be processed. Although a count-based window in which the update interval is per transaction still processes each expiring/incoming transaction at real time, processing each transaction against the entire stream in most cases is less efficient than processing a batch of transactions against the entire stream. In general, batch processing is more suitable for high-speed data streams. We note that *batch processing* here refers to processing a batch of transactions at a time and then using the mining results of the batch to update the results of the entire window seen so far, while we refer to the process of updating each transaction *against the entire window* as *tuple processing.* Thus, in Table 3, Manku and Motwani's [34] and Li et al.'s [31] algorithms actually tuple process, although their update interval is per batch.

5.2 Approximation Type, Support Error Bound and False Results

All the approximate algorithms adopt false-positive approaches [34, 31, 9, 21, 29, 10, 11], except for Yu et al.'s [45], which adopts a false-negative approach.

The false-positive approaches use a relaxed minimum support threshold, ϵ, to reduce the number of false-positive results and to obtain a more accurate frequency of the result itemsets. However, a larger set of sub-FIs will have to be kept for a smaller ϵ. The false-negative approach also uses a relaxed minimum support threshold, ϵ; however, the use of this ϵ lowers the probability of discarding an infrequent itemset that may become frequent later. Although a more accurate answer also requires a smaller ϵ, which generates more sub-FIs, the false-negative approach uses a second relaxed minimum support threshold, ϵ', that is increased gradually over time. This threshold, ϵ', restricts the sub-FIs to be kept in memory to only those whose support is no less than ϵ'. When a sufficient number of transactions have been received in the stream, ϵ' approaches σ and hence the number of sub-FIs kept is greatly reduced.

The error bound in the computed support and the possible false mining results of most of the false-positive approaches [34, 31, 21, 11] are implied by The Lossy Counting algorithm [34]. The false-negative approach [45] claims that the computed support is the same as the actual support with probability of at least $(1 - \delta)$ and each itemset in the exact mining result is outputted with probability of at least $(1 - \delta)$. We are not able to give unified equations to describe the support error bound and false mining results in Chang and Lee's [9, 10] and Lee and Lee's [29] algorithms. We refer the reader to the respective sections for the details.

Chi et al. [17] compute exact mining results over a sliding window. However, exact mining requires a huge amount of memory and CPU resources to keep track of all itemsets in the window and their actual frequency, even with a compact data structure as the one used in [17]. Mining FCIs may reduce the memory consumption since the set of FCIs is in general smaller than the set of FIs, but it still depends on the characteristics of the data stream.

5.3 Empirical Analysis

We analyze the performance of the algorithms according to the empirical studies in the respective papers that propose the algorithms.

Li et al. [31] compare the performance of their DSM-FI algorithm to that of the Lossy Counting algorithm by Manku and Motwani [34] on a synthetic dataset. According to their experiments, DSM-FI is more efficient than Lossy Counting by up to an order of magnitude and consumes less memory. Moreover, both the processing time and memory consumption of DSM-FI are more stable than those of Lossy Counting.

Yu et al. [45] also compare their FDPM algorithm to Lossy Counting. With an increase in the minimum support threshold σ, both the CPU time and the memory consumption of FDPM becomes significantly less than those of Lossy Counting. FDPM achieves high recall of over 95% and 100% precision, while Lossy Counting has 100% recall and at least 46% precision. The performance of FDPM is very stable for varying ϵ, while that of Lossy Counting is greatly influenced by ϵ. For small ϵ, Lossy Counting achieves high precision, but consumes more memory and is slower, while for large ϵ, Lossy Counting uses small memory and runs faster, at the expense of lowered precision.

In [29], Lee and Lee compare their algorithm estDec+ with Chang and Lee's algorithm estDec [9]. Since estDec+ mines FMIs, the processing time of estDec+ is considerably smaller than that of estDec, which mines FIs. The memory consumption of estDec+ is also considerably smaller than that of estDec due to the use of a compressed prefix tree structure in estDec+.

The landmark window model that Giannella et al. [21] adopt is specialized in the context (i.e. tilted-time window), thus their algorithm is not comparable to the others. However, their experiments record high throughput and low memory consumption of their algorithm.

Chi et al. [17] mine FCIs instead of FIs; their algorithm is thus not comparable with those algorithms that mine FIs. However, they compare their performance with the mine-from-scratch approach, that is, using the FCI mining algorithm CHARM [47] to mine the set of FCIs for each sliding window. Their results show that their algorithm outperforms the mine-from-scratch approach by over an order of magnitude.

Of the three algorithms of sliding window model, the two proposed by Chang and Lee [10, 11] only report the memory consumption in terms of the number of itemsets maintained and no comparison with other algorithms is provided. Thus, it is difficult for us to judge the performance of these algorithms.

6 Other Issues of Mining Data Streams

We first discuss in Sections 6.1 to 6.3 three important issues for the improvement of mining high-speed data streams. Then, we discuss the related work in Section 6.4.

6.1 Exact Mining Vs Approximate Mining

Exact mining requires keeping track of all itemsets in the window and their actual frequency, because any infrequent itemset may become frequent later in the stream. However, the number of all itemsets is $\mathcal{O}(2^{|\mathcal{I}|})$, making exact mining computationally intractable, in terms of both CPU and memory. Although exact mining may still be applicable for small sliding windows and for streams with relatively low data arrival rates, it is more realistic to compute approximate mining results over landmark windows or large sliding windows. In fact, approximation is a widely adopted technique in most of the stream-processing applications [20]. In the context of data mining, in most cases the goal is to identify generic, interesting or "out-of-the-ordinary" patterns rather than to provide results that are exact to the last decimal. Thus, approximate but instant answers, with reasonable accuracy, are particularly well-suited.

Approximating a set of FIs/FMIs/FCIs over a stream involves estimating both the set of resulting itemsets and their frequency. In the existing false-positive approaches [34, 31, 9, 21, 10, 11, 29], the set of sub-FIs kept is often too large in order to obtain a highly accurate answer. Thus, throughput is decreased and memory consumption is increased due to the processing of a large number of sub-FIs. A possible solution to this problem is first to use a *constant* lowered minimum support threshold to compute a set of potential FIs and then to use a *gradually increasing* lowered minimum support threshold to control the total number of sub-FIs kept in memory, as do Yu et al. [45]. We may consider the characteristics of different data streams to design specific (non-decreasing) functions to monitor the minimum support threshold at different points of a stream, thereby more effectively improving the mining accuracy and efficiency. It is also possible to design approximate algorithms that are neither false-positive nor false-negative but achieve both high recall and precision.

6.2 Load Shedding in Processing High-Speed Data Streams

Data streams are push-based and data arrival rates are high and often bursty. When the load of mining exceeds the capacity of the system, load shedding is needed to keep up with the arrival rates of the input streams. Several issues need to be considered when applying load shedding to process mining on data streams. First, we need to approximate the processing rate; that is, the number of transactions per unit of time that the system is able to handle. Thus, we need to consider the characteristics of a stream, such as the average size of a transaction and of an FI, as well as the memory requirement at the particular processing rate. Then, when the arrival rate of a stream is higher than the processing rate, we need to determine a technique to shed the load, such as random sampling [41, 48] or semantic drop and window reduction [40] or selective processing (i.e., processing only certain portions of the window). For the load shedding technique chosen, we also need to analyze the quality of the approximate mining results obtained.

6.3 Different Types of FIs

The exploratory nature of FI mining often results in a large number of FIs generated and handling such a large number of itemsets severely degrades the mining efficiency. A possible solution to reduce the number of itemsets mined is to mine FCIs [37] instead of FIs. The set of FCIs is a *complete* and *non-redundant* representation of the set of FIs [46] and the former is often orders of magnitude smaller than the latter. This significant reduction in the size of the result set leads to faster speed and less memory consumption in computing FCIs than in computing FIs. In addition, the set of FCIs also facilitates the generation of a set of non-redundant association rules [46].

Although many efficient algorithms [47, 42] on mining FCIs on static databases have been proposed, the only algorithm on mining FCIs (at the time of writing) is the Moment algorithm for data streams proposed by Chi et al. [17]. However, since Moment computes the exact set of FCIs and processes every transaction against the entire window, it is not efficient enough for handling high-speed streams and it consumes too much memory when the window becomes large. Therefore, a study of mining an approximate set of FCIs over a sliding window with reasonable guarantees on the quality of the approximation would be worthwhile. However, we note that it may not be feasible to mine FCIs over a landmark window since the number of FCIs approaches that of FIs when the window becomes very large.

When the number of FCIs is still too large, we may consider to mine FMIs [29]. Since the number of FMIs is orders of magnitude smaller than that of FIs and FCIs in most cases, it has been shown [24] that mining MFIs is significantly more efficient, in terms of both CPU and memory, than mining FIs and FCIs. However, a problem of FMIs is that FMIs lose the frequency information of their subset FIs and hence the error bound of the estimated frequency of the FIs recovered from the FMIs will be significantly increased. In mining static databases, many concise representations of FIs [8, 6, 39, 44, 5, 15] have been proposed. These concise representations of FIs are not only significantly smaller in size, but also able to recover the set of FIs with highly accurate frequency. Thus, it is also possible to consider mining these concise representations of FIs over the data streams. A challenge of mining these concise representations is how to efficiently update the concise set of FIs at the high arrival rate of the streaming data. Since each transaction may change the status of a large number of FIs (from concise FIs to normal FIs and vice versa), batch processing with efficient incremental update on a batch of transactions may be a better choice.

6.4 Other Related Work

A closely related work to mining FIs over streams is the mining of frequent items over data streams, which has been studied in a number of recent proposals [34, 19, 27, 12, 45, 33, 38]. However, mining FIs is far more challenging than counting singleton items due to the combinatorial explosion of the number of itemsets.

Extensive research has been conducted on querying processing and data management in data streams. The issues of data models, query languages in the context of data streams are the main research topics. We refer readers to the excellent surveys by Babcock et al. [4] and Golab and Özsu [23].

Some prior work on streaming algorithms for mining FIs include *Carma* [26] and *SWF* [30]. Carma scans a transaction database twice to mine the set of all FIs, where the first scan of Carma can be applied to compute a false-positive set of FIs over a landmark window. SWF mines the exact set of all FIs over a time-based sliding window. SWF first computes the set of all FIs over the first window and then it requires only one scan of the database for each incremental update. However, scanning the entire window for each slide may not be efficient enough to handle high-speed data streams.

For the recent development of FI/FCI mining over data streams, we are aware of the following two studies.

Jin and Agrawal [28] develop the *StreamMining Algorithm* using potential frequent 2-itemsets and the *Apriori* property to reduce the number of candidate itemsets. They also design a memory-resident summary data structure that implements a compact prefix tree using a hash table. Their algorithm is approximate and false-positive, which has deterministic bounds on the accuracy. The window model they adopt is the landmark window and they do not distinguish recent data from old data.

Cheng et al. [14] adopt a sliding window model to approximately mine FIs over data streams. The main contribution of this work is that it proposes a progressively increasing minimum support function to address the dilemma caused by ϵ in most of the existing algorithms. When an itemset is retained in the window longer, the technqiue requires its minimum support threshold to approach the minimum support of an FI. As a result, it is able to increase ϵ to significantly improve the mining efficiency and save memory consumption, at the expense of only slightly degraded accuracy. The proposed algorithm of this work, called *MineSW*, is shown by Cheng et al. to outperform an improved version of the algorithm proposed in [11] that apply Lossy Counting [34] algorithm to mine FIs over a sliding window.

7 Conclusions

In this paper, we provide a survey of research on mining data streams. We focus on frequent itemset mining and have tried to cover both early and recent literature related to mining frequent itemsets (FIs) or frequent closed itemsets (FCIs). In particular, we have discussed in detail a number of the state-of-the-art algorithms [34, 31, 45, 9, 21, 29, 10, 11, 17] on mining FIs, FMIs or FCIs over data streams. Moreover, we have addressed the merits and the limitations and presented an overall analysis of the algorithms, which can provide insights for end-users in applying or developing an appropriate algorithm for different streaming environments and various applications. We have also identified and discussed possible future research issues on mining algorithms.

We believe that as many new streaming applications and sensor network applications are becoming more mature and popular, streaming data and sensor data are also becoming richer. More high-speed data streams are generated in different application domains, such as millions of transactions generated from retail chains, millions of calls from telecommunication companies, millions of ATM and credit card operations processed by large banks, and millions of hits logged by popular Web sites. Mining techniques will then be very significant in order to conduct advanced analysis, such as determining trends and finding interesting patterns, on streaming data. It is our intention to present this survey to simulate interests in utilizing and developing the previous studies into emerging applications.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proc. of SIGMOD*, 1993.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of VLDB*, 1994.

[3] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proc. of IDCE*, 1995.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. of PODS*, 2002.

[5] F. Bonchi and C. Lucchese. On Condensed Representations of Constrained Frequent Patterns. In *KAIS*, 9(2): 180-201, 2005.

[6] J. F. Boulicaut, A. Bykowski and C. Rigotti. Free-Sets: a Condensed Representation of Boolean Data for the Approximation of Frequency Queries. In *DMKD*, 7(1):5-22, 2003.

[7] S. Brin, R. Motwani, and C. Silverstein. Beyond Market Basket: Generalizing Association Rules to Correlations. In *Proc. of SIGMOD*, 1997.

[8] T. Calders and B. Goethals. Mining All Non-derivable Frequent Itemsets. In *Proc. of PKDD*, 2002.

[9] J. H. Chang and W. S. Lee. Finding Recent Frequent Itemsets Adaptively over Online Data Streams. In *Proc. of KDD*, 2003.

[10] J. H. Chang and W. S. Lee. estWin: Adaptively Monitoring the Recent Change of Frequent Itemsets over Online Data Streams. In *Proc. of CIKM*, 2003.

[11] J. H. Chang and W. S. Lee. A Sliding Window method for Finding Recently Frequent Itemsets over Online Data Streams. In *Journal of Information Science and Engineering*, Vol. 20, No. 4, July, 2004.

[12] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Theor. Comput. Sci.*, 312(1): 3-15, 2004.

[13] Y. Chen, G. Dong, J. Han, B.W. Wah, and J. Wang. Multidimensional Regression Analysis of Time-Series Data Streams. In *Proc. of VLDB*, 2002.

[14] J. Cheng, Y. Ke, and W. Ng. Maintaining Frequent Itemsets over High-Speed Data Streams. In *Proc. of PAKDD*, 2006.

[15] J. Cheng, Y. Ke, and W. Ng. δ-Tolerance Closed Frequent Itemsets. To appear in *Proc. of ICDM*, 2006.

[16] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *The Annals of Mathematical Statistics*, 23(4):493-507, 1952.

[17] Y. Chi, H. Wang, P. S. Yu and R. R. Muntz. Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window. In *Proc. of ICDM*, 2004.

[18] Y. Chi, H. Wang, P. S. Yu and R. R. Muntz. Catch the Moment: Maintaining Closed Frequent Itemsets over a Data Stream Sliding Window. In *KAIS*, 10(3): 265-294, 2006.

[19] G. Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. In *Proc. of PODS*, 2003.

[20] M. Garofalakis, J. Gehrke, R. Rastogi. Querying and Mining Data Streams: You Only Get One Look. In *Tutorial of SIGMOD*, 2002.

[21] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In *Kargupta et al.: Data Mining: Next Generation Challenges and Future Directions*, MIT/AAAI Press, 2004.

[22] B. Goethals and M. Zaki. FIMI '03, Frequent Itemset Mining Implementations. In *Proc. of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.

[23] L. Golab and M. T. Özsu. Issues in Data Stream Management. In *SIGMOD Record*, 32(2): 5-14, 2003.

[24] K. Gouda and M. Zaki. Efficiently Mining Maximal Frequent Itemsets. In *Proc. of ICDM*, 2001.

[25] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. of SIGMOD*, 2000.

[26] C. Hidber. Online Association Rule Mining. In *Proc. of SIGMOD*, 1999.

[27] C. Jin, W. Qian, C. Sha, J. Yu, and A. Zhou. Dynamically Maintaining Frequent Items over a Data Stream. In *Proc. of CIKM*, 2003.

[28] R. Jin and G. Agrawal. An Algorithm for In-Core Frequent Itemset Mining on Streaming Data. In *Proc. of ICDM*, 2005.

[29] D. Lee and W. Lee. Finding Maximal Frequent Itemsets over Online Data Streams Adaptively. In *Proc. of ICDM*, 2005.

[30] C. Lee, C. Lin, and M. Chen. Sliding-window Filtering: an Efficient Algorithm for Incremental Mining. In *Proc. of CIKM*, 2001.

[31] H. Li, S. Lee, and M. Shan. An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams. In *Proc. of First International Workshop on Knowledge Discovery in Data Streams*, 2004.

[32] B. Liu, W. Hsu, and Y. Ma. Integrating Classification and Association Rule Mining. In *Proc. of KDD*, 1998.

[33] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *Proc. ICDE*, 2005.

[34] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. of VLDB*, 2002.

[35] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of Frequent Episodes in Event Sequences. In *DMKD*, 1:259-289, 1997.

[36] E. Omiecinski. Alternative Interest Measures for Mining Associations. In *IEEE TKDE*, 15:57-69, 2003.

[37] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering Frequent Closed Itemsets for Association Rules. In *Proc. of ICDT*, 1999.

[38] A. Pavan and S. Tirthapura. Range Efficient Computation of F0 over Massive Data Streams. In *Proc. ICDE*, 2005.

[39] J. Pei, G. Dong, W. Zou, and J. Han. Mining Condensed Frequent-Pattern Bases. In *KAIS*, 6(5): 570-594, 2004.

[40] U. Srivastava and J. Widom. Memory-Limited Execution of Windowed Stream Joins. In *Proc. of VLDB*, 2004.

[41] H. Toivonen. Sampling Large Databases for Association Rules. In *Proc. of VLDB*, 1996.

[42] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets. In *Proc. of KDD*, 2003.

[43] H. Wang, J. Yang, W. Wang, and P. S. Yu. Clustering by Pattern Similarity in Large Datasets. In *Proc. of SIGMOD*, 2002.

[44] D. Xin, J. Han, X. Yan, and H. Cheng. Mining Compressed Frequent-Pattern Sets. In *Proc. of VLDB*, 2005.

[45] J. Yu, Z. Chong, H. Lu, and A. Zhou. False Positive or False Negative: Mining Frequent Itemsets from High Speed Transactional Data Streams. In *Proc. of VLDB*, 2004.

[46] M. Zaki. Generating Non-Redundant Association Rules. In *Proc. of KDD*, 2000.

[47] M. Zaki and C. J. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. In *Proc. of SDM*, 2002.

[48] M. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of Sampling for Data Mining of Association Rules. In *Proc. of RIDE*, 1997.