J Intell Inf Syst (2008) 31:191–215
DOI 10.1007/s10844-007-0042-3

Maintaining frequent closed itemsets over a sliding window

James Cheng · Yiping Ke · Wilfred Ng

Received: 2 October 2006 / Revised: 13 April 2007 /
Accepted: 13 April 2007 / Published online: 20 June 2007
© Springer Science + Business Media, LLC 2007

Abstract In this paper, we study the incremental update of *Frequent Closed Itemsets* (*FCIs*) over a sliding window in a high-speed data stream. We propose *the notion of semi-FCIs*, which is to progressively increase the minimum support threshold for an itemset as it is retained longer in the window, thereby drastically reducing the number of itemsets that need to be maintained and processed. We explore the properties of semi-FCIs and observe that a majority of the subsets of a semi-FCI are not semi-FCIs and need not be updated. This finding allows us to devise an efficient algorithm, *IncMine*, that incrementally updates the set of semi-FCIs over a sliding window. We also develop an *inverted index* to facilitate the update process. Our empirical results show that IncMine achieves significantly higher throughput and consumes less memory than the state-of-the-art streaming algorithms for mining FCIs and FIs. IncMine also attains high accuracy of 100% precision and over 93% recall.

Keywords Frequent Closed Itemset · Data stream mining · Sliding window

1 Introduction

Recently, the increasing prominence of data streams has led to the study of online mining of *Frequent Itemsets* (*FIs*), which is an important mining task for a wide range

J. Cheng (✉) · Y. Ke · W. Ng
Department of Computer Science and Engineering,
The Hong Kong University of Science and Technology,
Clear Water Bay, Hong Kong
e-mail: csjames@cse.ust.hk

Y. Ke
e-mail: keyiping@cse.ust.hk

W. Ng
e-mail: wilfred@cse.ust.hk

of applications (Garofalakis et al. 2002), such as web log and click-stream mining, network traffic analysis, trend analysis and fraud/anomaly detection in telecom data, e-business, stock market analysis, and sensor networks. With the rapid emergence of these new application domains, it has become increasingly demanding to discover interesting trends, patterns and exceptions over high-speed data streams.

Mining FIs (Agrawal et al. 1993) is fundamental to many important data mining tasks, such as associations and correlations, and has been studied extensively with static datasets. However, mining data streams poses many new challenges. First, it is unrealistic to keep the entire stream in main memory or even in secondary storage, since a data stream comes continuously and the amount of data is unbounded. Second, traditional methods of mining over 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 very short response time. In addition, the combinatorial explosion of itemsets exacerbates stream mining in terms of both memory consumption and processing efficiency.

Previous work (Manku and Motwani 2002; Li et al. 2004; Yu et al. 2004) on stream mining has studied mining an *approximate* set of FIs, with an error bound, over the entire history of a stream without distinguishing recent itemsets from old ones. However, the importance of an itemset in a stream usually decreases with time and many applications only focus on the most recent patterns. For example, the detection of network intrusions is based on changes of the frequent patterns in the prior few minutes. Therefore, other existing work has placed greater importance on recent data by either adopting a sliding window model (Lee et al. 2001; Chang and Lee 2004; Chi et al. 2004; Jiang and Gruenwald 2006) or discounting the importance of old itemsets exponentially as time goes on (Chang and Lee 2003; Giannella et al. 2004).

All the above-mentioned work, except Chi et al. (2004) and Jiang and Gruenwald (2006), focuses on mining FIs. However, the set of FIs is often too large for the mining to be efficient. In Pasquier et al. (1999), the notion of *Frequent Closed Itemsets* (*FCIs*), which are FIs that have no proper superset with the same *support* (i.e., occurrence frequency), is introduced. The set of FCIs is a *complete* and *non-redundant* representation of the set of FIs and, as reported by Zaki (2000), 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 mining FCIs instead of FIs. In addition, the set of FCIs also enables the generation of non-redundant association rules (Zaki 2000).

In this paper, we study the problem of incrementally updating the set of FCIs over a sliding window. The sliding window is composed of a sequence of time units. Each time unit receives a variable number of transactions, from which a set of *local* FCIs is computed and then used to update the set of *global* FCIs over the entire sliding window. We highlight the main issues that we address in this paper as follows.

First, in the stream setting, an infrequent itemset may become frequent later on. To avoid wrongly discarding such an itemset, previous methods (Manku and Motwani 2002; Chang and Lee 2003; Giannella et al. 2004; Li et al. 2004; Chang and Lee 2004) use a *relaxed minimum support threshold* to keep an extra set of infrequent itemsets that have a high potential to become frequent later. However, the size of this extra set is often too large if we want to obtain more accurate answers. We propose *the notion of semi-FCIs*, which is to progressively increase the minimum

support threshold for an itemset as it is retained longer in the window. If an itemset has low support when it first arrives in the stream, we will require its support to become higher later to compensate for its low support observed earlier. Thus, we do not use the relaxed minimum support threshold throughout the stream as do in the previous studies. This allows us to effectively identify and drop the unpromising itemsets, thereby drastically reducing the number of itemsets that need to be kept and processed.

Second, we propose a novel algorithm, called *IncMine*, to incrementally update a set of semi-FCIs over a sliding window. Since an FCI may become non-closed and an FI may become an FCI as the window slides, we essentially need to keep track of all FIs. This is inefficient since the number of FIs is significantly larger than that of FCIs, as evidenced in Pasquier et al. (1999); Zaki (2000); Zaki and Hsiao (2002) and Wang et al. (2003). We explore the properties of semi-FCIs and find that the status of an itemset can be determined by simply investigating the relationship between the semi-FCIs over two consecutive slides. This important finding also indicates that only a small portion of the subsets of a semi-FCI in the current window will become semi-FCIs in the window at the next slide; as a consequence, the majority of the FIs do not need to be updated. Based on this, we devise our algorithm IncMine, which efficiently performs the incremental update of semi-FCIs by pruning a large portion of the subsets of a semi-FCI that will not be semi-FCIs.

Third, we design an *inverted index* to facilitate the update process. We show that the prefix tree structure, which is commonly used in mining both FIs and FCIs, is not efficient in processing some update operations such as the search for a semi-FCI that is the smallest proper superset of an itemset. We therefore propose an inverted index structure to store the semi-FCIs and we show that the inverted index not only supports efficient processing of the update operations, but is also space efficient.

Finally, we evaluate the performance of IncMine by extensive experiments. The results show that IncMine achieves throughput that is up to orders of magnitude higher than the state-of-the-art stream mining algorithms, *Moment* (Chi et al. 2004) and a variant of Lossy Counting (Manku and Motwani 2002; Chang and Lee 2004) that incrementally updates a set of FIs over a sliding window. IncMine also consumes significantly less memory than do the other algorithms. Although IncMine computes approximate answers, we show that IncMine attains very high accuracy of 100% precision and over 93% recall.

Organization This paper is organized as follows. Section 2 discusses the related work and Section 3 gives the preliminaries of mining FCIs over a sliding window. Section 4 presents the notion of semi-FCIs. Section 5 discusses the algorithm IncMine. Section 6 describes the inverted index structure. Section 7 reports the experimental results and Section 8 concludes the paper.

2 Related work

We limit our discussion to mining FIs and FCIs over data stream. To our knowledge, mining FCIs over data streams is only studied by Chi et al. (2004) and Jiang and Gruenwald (2006). Their methods incrementally update the set of FCIs over a sliding

window whenever a new transaction comes into or an old one leaves the window. This update-per-transaction is not efficient for handling streams with a high arrival rate, as shown by our experiments. However, their methods compute the *exact* set of FCIs, which is a desirable feature to applications requiring accurate answers. In contrast, our objective is to obtain high throughput for high-speed streams, at the expense of slightly lowered accuracy.

Other existing work mainly focuses on mining FIs. Approaches that also favor recent data to older data include (Lee et al. 2001; Chang and Lee 2004, 2003; Giannella et al. 2004). Lee et al. (2001) compute the exact set of FIs over a sliding window. Their method scans the entire window at each slide and is more suitable for offline processing. Chang and Lee (2004) adopt the sliding window model to mine recent FIs based on the estimation mechanism of *Lossy Counting* (Manku and Motwani 2002). Chang and Lee (2003) also develop an approximate algorithm that uses a decay rate to diminish exponentially the effect of old transactions on the mining result. Giannella et al. (2004) adopt a tilted-time window model (Chen et al. 2002) to incrementally update an approximate set of FIs at multiple time granularities to answer time-sensitive queries.

Other approaches (Manku and Motwani 2002; Li et al. 2004; Yu et al. 2004) aim at mining FIs over the entire history of a stream. Manku and Motwani (2002) propose to approximate the set of FIs and use a user-specified error threshold to control the quality of the approximation. Li et al. (2004) design a prefix-tree-based data structure that computes an approximate set of FIs with bounded memory usage. Yu et al. (2004) adopt the *Chernoff bound* to develop false-negative oriented mining algorithms that can control the bound of memory usage and the quality of the approximation by predefined parameters.

3 Preliminaries

Let $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of items. An *itemset* (or a *pattern*) is a subset of $\mathcal{I}$. A *transaction*, X, is an itemset and X *supports* an itemset, Y, if $X \supseteq Y$. A *transaction data stream* is a sequence of incoming transactions. We denote a *time unit* in the stream as t_i, within which a variable number of transactions may arrive. A *window* or a *time interval* in the stream is a set of successive time units, denoted as $T = \langle t_i, \ldots, t_j \rangle$, where $i \leq j$, or simply $T = t_i$ if $i = j$. A *sliding window* in the stream is a window of a fixed number of time units that slides forward for every time unit. In this paper, we use t_τ to denote the *current time unit*. Thus, the *current window* is $W = \langle t_{\tau-w+1}, \ldots, t_\tau \rangle$, where w denotes the *size* of W, i.e., the number of time units in W.

We define $trans(T)$ as the set of transactions that arrive on the stream in a time interval T and $|trans(T)|$ as the number of transactions in $trans(T)$. The *support* of an itemset X over T, denoted as $sup(X, T)$, is the number of transactions in $trans(T)$ that support X. Given a predefined *Minimum Support Threshold* (*MST*), σ $(0 < \sigma \leq 1)$, we say that X is an *FI* over T if $sup(X, T) \geq \sigma |trans(T)|$. X is a *Frequent Closed Itemset* (*FCI*) over T if X is an FI over T and there exists no Y such that $Y \supset X$ and $sup(Y, T) = sup(X, T)$. If $X \supset Z$ and $sup(X, T) = sup(Z, T)$, where X is an FCI over T, we denote the relationship between X and Z as $X \sqsupset^T Z$, or equivalently, $Z \sqsubset^T X$.

Table 1 Transactions in a stream of three time units

t_1	t_2	t_3
abcd	abc	abcd
abcx	abcd	az
bcy		abc

Given a transaction data stream and an MST, the problem of *FCI mining over a sliding window in the stream* is *to find the set of all FCIs over the window at each slide*.

Example 1 Table 1 records the transactions that arrive in the stream in two successive windows, $W_1 = \langle t_1, t_2 \rangle$ and $W_2 = \langle t_2, t_3 \rangle$. For brevity, an itemset {a, b, c, d} is written as abcd. If the minimum support required is 2 (i.e., $\sigma = \frac{2}{5}$ in both windows), then the set of FCIs over W_1 and W_2 are {abcd, abc, bc} and {abcd, abc, a}, respectively. For example, bc is an FCI over W_1 since $sup(\texttt{bc}, W_1) = 5$ and no proper superset of bc has the same support as bc over W_1; however, bc is not an FCI over W_2 since $\texttt{bc} \sqsubset^{W_2} \texttt{abc}$. Also note that there are 15 FIs over both windows, that is, all non-empty subsets of abcd.

4 Semi-frequent closed itemsets

An itemset may be infrequent at some point in a stream but becomes frequent later. Since there are exponentially many infrequent itemsets at any point in a stream, it is infeasible to keep all infrequent itemsets. Suppose we have an itemset X which becomes frequent after time t. Since X is infrequent before t, the support of X in the stream before t is lost. A common approach (Manku and Motwani 2002; Li et al. 2004; Chang and Lee 2004) to estimate X's support before t is to use an *error parameter*, ϵ, where $0 \leq \epsilon \leq \sigma$. X is maintained in the window as long as its support is at least ϵN, where N is the number of transactions in the current window. Thus, if X is kept only after t, the support of X before t is at most ϵN. However, the use of ϵ leads to a dilemma. A small ϵ gives an estimated support close to the true support. Unfortunately, a small ϵ also results in a large number of itemsets to be processed and maintained, thereby drastically increasing the memory consumption and severely degrading the processing efficiency. To tackle this problem, we consider ϵ as a *relaxed MST* and propose to *progressively increase* the value of ϵ for an itemset as it is retained longer in the window.

We use the relaxed MST $\epsilon = r\sigma$, where r $(0 \leq r \leq 1)$ is the *relaxation rate*, to mine FCIs over each time unit t in the sliding window. Since all itemsets whose support is less than $r\sigma|trans(t)|$ are discarded, we define the *approximate support* of an itemset as follows.

Definition 1 (Approximate support) The *approximate support* of an itemset X over a time unit t is defined as

$$\widetilde{sup}(X, t) = \begin{cases} 0 & \text{if } sup(X, t) < r\sigma|trans(t)| \\ sup(X, t) & \text{otherwise.} \end{cases}$$

The approximate support of X over a time interval $T = \langle t_j, \ldots, t_k \rangle$ is defined as

$$\widetilde{sup}(X, T) = \sum_{i=j}^{k} \widetilde{sup}(X, t_i).$$

Based on the approximate support of an itemset, we apply *a progressively increasing MST function* to define a *semi-frequent closed itemset*.

Definition 2 (Semi-frequent closed itemset) Let $W = \langle t_{\tau-w+1}, \ldots, t_\tau \rangle$ be a sliding window of size w and $T^k = \langle t_{\tau-k+1}, \ldots, t_\tau \rangle$ be the most recent k time units in W, where $1 \leq k \leq w$.

Given a *non-decreasing* function, $minsup(k)$, where $\forall k \in \{1, \ldots, w\}$, $0 \leq minsup(k) \leq \sigma |trans(T^k)|$, an itemset X is a *semi-FI* over W if $\exists k$ such that $\widetilde{sup}(X, T^k) \geq minsup(k)$. X is a *semi-frequent closed itemset* (*semi-FCI*) over W if X is a semi-FI and $\nexists Y \supset X$ such that $\widetilde{sup}(Y, T^k) = \widetilde{sup}(X, T^k)$. X is called a k-semi-FCI if X is a semi-FCI and k is given by $MAX\{k : \widetilde{sup}(X, T^k) \geq minsup(k)\}$.

Let X be a k-semi-FCI. If $X \supset Y$ and $\widetilde{sup}(X, T^k) = \widetilde{sup}(Y, T^k)$, we denote the relationship between X and Y as $X \sqsupset^W Y$ or $Y \sqsubset^W X$ (or more precisely, $X \sqsupset^{T^k} Y$ or $Y \sqsubset^{T^k} X$).

Definition 3 (MST function) We define a *progressively increasing MST function*, $minsup(k)$, as follows:

$$minsup(k) = \left\lceil m_k \times r_k \right\rceil,$$

where $m_k = \sigma \left| trans\left(T^k\right) \right|$ and $r_k = \left(\frac{1-r}{w}\right)\left(k-1\right) + r$.

The term m_k in the function $minsup(k)$ in Definition 3 is the minimum support required for an FI over T^k, while the term r_k progressively increases the relaxed MST at the rate of $(\frac{1-r}{w})$ for each older time unit in the window. We keep an itemset in the window only if the approximate support of the itemset over T^k is no less than $minsup(k)$ for some $k \in \{1, \ldots, w\}$. Meanwhile, for each semi-FCI X, we always compute the maximum k at which X is a k-semi-FCI.

We remark that k is relative to the specific window at each slide and the approximate support of a semi-FCI X over $\langle t_{\tau-w+1}, \ldots, t_{\tau-k} \rangle$ is considered as unpromising and discarded. When we re-compute the value of k for the window at the next slide, the approximate support of X over $\langle t_{\tau-w+1}, \ldots, t_{\tau-k} \rangle$ will be taken as 0.

We illustrate the concept of semi-FCIs by the following example.

Example 2 Let $\sigma = 0.01$, $r = 0.1$ and $w = 10$. We assume a uniform input rate of 2,000 transactions in each time unit. We consider 11 time units that constitute to the windows for the following two slides, $W_1 = \langle t_1, \ldots, t_{10} \rangle$ and $W_2 = \langle t_2, \ldots, t_{11} \rangle$. Table 2 shows the value of $minsup(k)$, for $1 \leq k \leq 10$, and the support of two itemsets, `ab` and `cd`, over each time unit.

We first discuss the method used by the state-of-the-art algorithm *Lossy Counting* (Manku and Motwani 2002; Chang and Lee 2004), which is the baseline of comparison with our approach in our experiments. Once an itemset is discovered as frequent

Table 2 Two sample semi-FCIs

k	10	9	8	7	6	5	4	3	2	1	
$minsup(k)$	182	148	117	90	66	46	30	17	8	2	
Time Unit	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
$sup(\texttt{ab},t_i)$	3	1	2	3	2	1	4	7	11	19	21
$sup(\texttt{cd},t_i)$	3	11	20	29	11	8	17	28	37	41	39

over some time unit t_i, Lossy Counting keeps the itemset as far as its overall support is no less than $r\sigma|trans(\langle t_i, \ldots, t_\tau\rangle)|$. Since $r\sigma|trans(W_1)| = r\sigma|trans(W_2)| = 20$, which is small, both ab and cd are retained in W_1 and W_2, even though the support of ab over some time units is very low. In particular, the support of ab is 1 over t_2 and t_6 but ab is still mined. Mining itemsets with minimum support 1 means mining all itemsets, which is an extremely expensive operation, since the number of all itemsets is prohibitively large. Although we only need to consider those itemsets that have already been discovered, the number of such itemsets is still very large when σ is small.

With the progressively increasing MST, the support of ab over any two consecutive time units within $\langle t_1, \ldots, t_6\rangle$ is always less than $minsup(2) = 8$. For ab to be frequent over W_1 or W_2, its support must increase rapidly from t_7 to t_{11}, which is unlikely given the low support of ab in the past. Thus, the support of ab over $\langle t_1, \ldots, t_6\rangle$ are regarded as *unpromising* and discarded. Although the trend in the support of ab from t_7 to t_{11} shows that ab may become frequent after a few window slides, at that time the first few time units will have expired anyway. Therefore, it is reasonable that *we require the support of* ab *to increase progressively as it is retained longer in the window*. In this way, ab is only kept from t_7 and afterwards. Indeed, *the support of* ab *over these most recent time units is more likely to contribute to the overall support of* ab *should it become frequent later*.

The support of cd over all time units in both W_1 and W_2 is always greater than the corresponding $minsup(k)$, for $k = 1, \ldots, 10$. However, cd also has some low support over a few time units. *These low support records are not discarded since they are compensated with the high support of* cd *over other time units*.

4.1 Quality of approximation

We propose to incrementally update the set of all k-semi-FCIs over a sliding window and return those k-semi-FCIs whose approximate support is no less than $\sigma|trans(W)|$ as the mining result for the window W at each current slide. We analyze the quality of the approximation on the mining result returned.

The error bound of the approximate support of a k-semi-FCI X over T^k, $\widetilde{sup}(X, T^k)$, is described as follows:

$$(sup(X, T^k) - \mathcal{E}) \leq \widetilde{sup}(X, T^k) \leq sup(X, T^k), \tag{1}$$

where $\mathcal{E} = \sum_{i\in I}\left(r\sigma\left|trans(t_i)\right| - 1\right)$ and $I = \{i : (\tau - k + 1 \leq i \leq \tau) \wedge (\widetilde{sup}(X, t_i) = 0)\}$.

Note that $\widetilde{sup}(X, t_i) = 0$ implies that X is infrequent over t_i. Thus, the true support of X over t_i is at most $(r\sigma|trans(t_i)| - 1)$.

Replacing $r\sigma$ with ϵ and $|trans(T^k)|$ with N_k, we obtain an upper bound for the *maximum support error* $\mathcal{E}$ as follows:

$$\mathcal{E} < \epsilon N_k \tag{2}$$

In most cases (except for skewed data distribution which is addressed in Section 4.2), $|I|$ is small (otherwise, X would not satisfy the requirement of a k-semi-FCI). More importantly, the k for most k-semi-FCIs over a window is equal to w, implying that $|I|$ is even smaller since no unpromising support is discarded for a w-semi-FCI. Thus, in most cases, we have $\mathcal{E} \ll \epsilon N_k$. When $k = w$, we have $\mathcal{E} \ll \epsilon N$, where $N = |trans(W)|$. We remark that ϵN is the maximum support error bound of most existing approximate stream mining algorithms (Manku and Motwani 2002; Giannella et al. 2004; Li et al. 2004; Chang and Lee 2004). When $k < w$, $\mathcal{E}$ is in general larger since there is some loss in support over the time interval $\langle t_{\tau-w+1}, \ldots, t_{\tau-k}\rangle$. However, the support of the k-semi-FCI over this time interval is considered unpromising and is expiring [this is unlike in the landmark window (Manku and Motwani 2002) where the support of an itemset does not expire].

Our method is a false-negative approach (Yu et al. 2004). The set of false-negatives is defined as $\{X : (\widetilde{sup}(X, W) < \sigma|trans(W)|) \wedge (sup(X, W) \geq \sigma|trans(W)|)\}$. If it happens that $\widetilde{sup}(X, W) < \sigma|trans(W)|$ but $sup(X, W) \geq \sigma|trans(W)|$, then there exists some time unit t_i in W such that $\widetilde{sup}(X, t_i) = 0$. Recall that $\widetilde{sup}(X, t_i) = 0$ only if $sup(X, t_i) < \epsilon|trans(t_i)|$ or $\widetilde{sup}(X, t_i)$ is unpromising and discarded. Thus, the true support of X over other time units in W must be much greater such that we can still have $sup(X, W) \geq \sigma|trans(W)|$. Therefore, we can deduce that false-negatives are mostly itemsets with skewed support distribution over the stream, which can be addressed using a specific *minsup* function. In most other cases, the number of false-negatives is small as will be verified by our experiments in Section 7.

4.2 Characterizing the *minsup* function

Different date streams have different characteristics and the characteristics of a data stream may also vary over time. Thus, the *minsup* function described in Definition 3 may not fit every data stream. However, we can employ specific functions to fit specific data streams. For example, some patterns may occur less frequently at the beginning but much more frequently towards the end of their life time, e.g., data from an online auction; in this case, we can adopt a curve that rises gently at the beginning, but more steeply after a certain point in time, e.g., when the auction is about to close. We can explore the characteristics of data streams to design tailor-made *minsup* functions, as well as apply techniques of detecting changes in data streams (Kifer et al. 2004) to tune the *minsup* functions to accommodate with the changes in streams over time.

5 Incremental update of semi-FCIs

In this section, we investigate the properties of semi-FCIs and propose an efficient algorithm, *IncMine*, to incrementally update the set of semi-FCIs over a sliding window.

5.1 Problem statement

From now on, we let t_τ be the current time unit, $W_L = \langle t_{\tau-w}, \ldots, t_{\tau-1}\rangle$ and $W_C = \langle t_{\tau-w+1}, \ldots, t_\tau\rangle$ be the *last* window and the *current* window, respectively, as depicted in Fig. 1. We also let F, L and C be the set of semi-FCIs over t_τ, W_L and W_C, respectively. Thus, "$X \in F$ (similarly for L or C)" is equivalent to "X is a semi-FCI over t_τ (similarly for W_L or W_C)".

We also refer to the support and the approximate support of an itemset interchangeably in the rest of the paper.

The task of *incrementally updating the set of semi-FCIs over a sliding window* is *to update L with F to give C*, where F is generated with a relaxed MST $r\sigma$ by an existing non-streaming FCI mining algorithm (Pasquier et al. 1999; Zaki and Hsiao 2002; Wang et al. 2003). All k-semi-FCIs over W_C that have support no less than $\sigma|trans(W_C)|$ are then outputted as the result at each slide.

5.2 A naive update algorithm

We first describe a naive algorithm for performing the incremental update of semi-FCIs, which is shown in Algorithm 1.

Lines 1-8 of Algorithm 1 first update the support of each semi-FCI $Y \in F$ and the support of Y's subsets. Line 3 ensures that a subset X of Y is always updated with X's smallest superset in F, since only this superset has the same support as X. For example, if ab and abc are both in F and a is not in F, then $\widetilde{sup}(\mathtt{a},t_\tau) = \widetilde{sup}(\mathtt{ab},t_\tau) > \widetilde{sup}(\mathtt{abc},t_\tau)$ and thus a is not updated with abc but with ab. After the support of X over t_τ is correctly updated (Line 4), if X is not in L, we add X to L and retrieve its support over W_L, except that over the expired time unit $t_{\tau-w}$, from X's smallest superset in L, if any, which has the same support as X over W_L (Lines 5–8).

After the support of all the itemsets has been updated, Lines 9–13 of the algorithm verify whether an itemset X satisfies the requirement of being a semi-FCI. X is removed from L if it is not a semi-FCI over W_C; otherwise, we keep X in L after discarding its unpromising support records (Line 11). Finally, we return the updated L as C (Line 14).

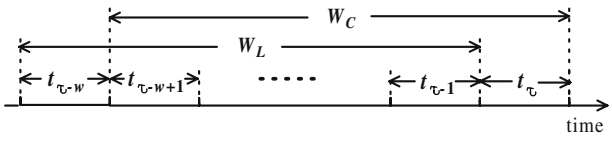


Fig. 1 A sliding window

Algorithm 1 *NaiveUpdate*(F,L)

1. **for each** $Y \in F$ **do**
2. **for each** $X \subseteq Y$ **do**
3. **if** $(\nexists Z \in F$ such that $X \subseteq Z \subset Y)$
4. $\widetilde{sup}(X, t_\tau) \leftarrow \widetilde{sup}(Y, t_\tau)$;
5. **if** $(X \notin L)$
6. $L \leftarrow L \cup \{X\}$;
7. **if** $(\exists Z \in L$ such that $Z \sqsupset^{W_L} X)$
8. $\widetilde{sup}(X, t_i) \leftarrow \widetilde{sup}(Z, t_i)$, for $\tau - w + 1 \leq i \leq \tau - 1$;
9. **for each** $X \in L$ **do**
10. $k \leftarrow MAX\{k : (1 \leq k \leq w) \wedge (\widetilde{sup}(X, T^k) \geq minsup(k))\}$;
11. delete $\widetilde{sup}(X, t_i), \forall i < \tau - k + 1$;
12. **if** $(\exists Z \in L$ such that $Z \supset X$ and $\widetilde{sup}(Z, T^k) = \widetilde{sup}(X, T^k))$
13. $L \leftarrow L - \{X\}$;
14. **return** $C \leftarrow L$;

5.3 Properties of semi-FCIs

Algorithm 1 exhaustively enumerates every subset of each itemset in F (Lines 1–3). This wastes a great amount of processing power since a large number of subsets of the semi-FCIs in F overlap. We show in the following lemma that we can avoid repeatedly processing these common subsets.

Lemma 1 *Let S be the set of FCIs over a time interval T. Given two FIs, X and Y, over T. If $Y \in S$ and $X \sqsubset^T Y$, then $\forall Z \in S$, if $X \subset Z$ and $Z \neq Y$, then $Y \subset Z$.*

Proof Suppose to the contrary that $Y \not\subset Z$. By the assumption that $X \subset Z$ and $Z \neq Y$, it follows that $\exists y$ such that $y \in (Y - Z)$ and $y \notin X$. From $X \sqsubset^T Y$, we have $sup(X, T) = sup(Y, T)$, which means that every transaction supporting X also supports Y (and y). $X \subset Z$ means that every transaction supporting Z also supports X and hence Y (and y). Thus, we have $Z' = Z \cup \{y\}$ in every transaction that supports Z, contradicting the assumption that Z is an FCI. Therefore, $Y \subset Z$. □

Lemma 1 is an application of Lemma 3.5 in Pei et al. (2000) in our context. Note that Lemma 1 can be directly applied to semi-FCIs, since it follows from Definition 1 that the support of an itemset over a window obeys the additive property of the support of the itemset over each time unit in the window.

Lemma 1 shows that for any semi-FI X that is not a semi-FCI over T, there is a unique semi-FCI Y such that $X \sqsubset^T Y$ and the size of Y is the smallest among all semi-FCIs that are supersets of X. We call Y the *Smallest semi-FCI Superset* (*SFS*) of X.

Recall that Lines 3-4 of Algorithm 1 make sure that an itemset X is updated with either its SFS or itself (if $X \in F$), since only X or its SFS has the same support as X over t_τ. The verification of SFS in Line 3 is costly since it scans F once for each subset of $Y \in F$. However, if we modify Lines 1-2 of Algorithm 1 to be Lines 1-2 of Algorithm 2, in which we order the semi-FCIs in F in ascending order of their

sizes and process a larger subset of Y before a smaller one, we can avoid the costly verification of SFS and the repeated processing of the common subsets of the semi-FCIs in F, as proved by Lemma 2.

Algorithm 2 *OrderedUpdate*(F,L)

```
1.  for each Y ∈ F in size-ascending order do
2.      for each X ⊆ Y in size-descending order do
3.          if (X is updated)
4.              skip X and all its subsets;
5.          else
6-16.           — Same as Lines 4-14 of Algorithm 1 —
```

Lemma 2 *In Algorithm 2, if X has been updated previously, then all subsets of X must have also been updated.*

Proof Line 1 of Algorithm 2 ensures that the SFS of an itemset is always ordered before all other supersets of the itemset. Thus, an itemset is always *first* updated with its SFS or itself. If a subset X of Y has been updated previously, then, by Lemma 1, X must be updated with some Z that is ordered before Y in F, and the subsets of X must also be updated with Z or some semi-FCI ordered before Z in F. □

For example, if `abc` and `abcd` are both in F, then `abc` (and all its subsets) must be processed before we process `abcd`. Thus, when we process the subsets of `abcd`, we can skip processing all itemsets that are subsets of `abc`.

Although Algorithm 2 avoids common subsets of the semi-FCIs in F being repeatedly processed, it still processes every distinct subset of the semi-FCIs at least once. The number of these distinct subsets is still very large, since an itemset of size n has $(2^n - 1)$ nonempty subsets. According to Zaki (2000), the number of FCIs is often orders of magnitude smaller than that of the corresponding FIs. Therefore, the majority of the $(2^n - 1)$ subsets of a semi-FCI are not semi-FCIs. Thus, a huge amount of memory and CPU power are used in processing the non-semi-FCI subsets. The following theorem formally states the relationships between the semi-FCIs in the sets F, L and C, and indicates that a large portion of the subsets of a semi-FCI in F will not be a semi-FCI in C and thus these subsets do not need to be updated.

Theorem 1 *Given a semi-FI, X, over t_τ,*

(a) *If $X \in F$, then $X \in C$.*
(b) *If $X \notin F$ and $X \in L$, then $X \in C$ if and only if $\exists Y \in F$ such that $X \sqsubset^{t_\tau} Y$; $\exists k \in \{2, \ldots, w\}$ such that $\widetilde{sup}(X, T^k) \geq minsup(k)$; and $\forall X'$, where $X \subset X' \subseteq Y$, $\widetilde{sup}(X, T^k) > \widetilde{sup}(X', T^k)$.*
(c) *If $X \notin F$ and $X \notin L$, then $X \in C$ if and only if $\exists Y \in F \backslash L$ such that $X \sqsubset^{t_\tau} Y$; $\exists Z \in L$ such that $X \sqsubset^{W_L} Z$ and $Y \cap Z = X$; and $\exists k \in \{2, \ldots, w\}$ such that $\widetilde{sup}(X, T^k) \geq minsup(k)$.*

Proof

(a) If $X \in F$, then X is at least a 1-semi-FCI over W_C. It thus follows that $X \in C$. In both Parts (b) and (c), we have $X \notin F$. Since X is a semi-FI over t_τ, it follows that $\exists Y \in F$ such that $X \sqsubset^{t_\tau} Y$, which further implies that X cannot be a 1-semi-FCI over W_C and hence $k \neq 1$.

(b) (*Only if*) $X \in C$ means that X is a k-semi-FCI over W_C for some $k \in \{2, \ldots, w\}$. By Definition 2, $\widetilde{sup}(X,T^k) \geq minsup(k)$ and $\forall X \subset X'$, $\widetilde{sup}(X, T^k) > \widetilde{sup}(X', T^k)$.
(*If*) Since $\widetilde{sup}(X,T^k) \geq minsup(k)$ for some $k \in \{2, \ldots, w\}$, it remains to prove that $\forall X' \supset X, \widetilde{sup}(X, T^k) > \widetilde{sup}(X', T^k)$. If $X \subset X' \subseteq Y$, we have $\widetilde{sup}(X,T^k) > \widetilde{sup}(X',T^k)$. Otherwise, if $X' \nsubseteq Y$, since $X \sqsubset^{t_\tau} Y$, we have $\widetilde{sup}(X, t_\tau) > \widetilde{sup}(X', t_\tau)$ and hence $\widetilde{sup}(X, T^k) > \widetilde{sup}(X', T^k)$. Therefore, $X \in C$ follows.

(c) (*Only if*) Since $X \notin L$ and $X \in C$, then X is a semi-FI over W_L, or otherwise, $\widetilde{sup}(X, W_L) = 0$, implying that $X \sqsubset^{W_C} Y$ and thus $X \notin C$, which leads to a contradiction. Therefore, $\exists Z \in L$ such that $X \sqsubset^{W_L} Z$.
Since $X \in C$, we have $\widetilde{sup}(X,T^k) \geq minsup(k)$, for some $k \in \{2, \ldots, w\}$.
We now prove $Y \notin L$. Suppose to the contrary that $Y \in L$. Since $X \sqsubset^{W_L} Z$, we have either $Y = Z$ or by Lemma 1, $X \subset Z \subset Y$. In either case, $\widetilde{sup}(X, T^k) = \sum_{i=\tau-k+1}^{\tau-1} \widetilde{sup}(Z, t_i) + \widetilde{sup}(Y, t_\tau) = \widetilde{sup}(Z, T^k)$, implying that $X \sqsubset^{W_C} Z$ and hence $X \notin C$, which leads to a contradiction.
Finally, we prove $(Y \cap Z) = X$. Since $X \sqsubset^{t_\tau} Y$ and $X \sqsubset^{W_L} Z$, we have $X \subset Y$ and $X \subset Z$, and hence $X \subset (Y \cap Z)$ or $X = (Y \cap Z)$. Suppose that $X \subset (Y \cap Z)$. Let $X' = (Y \cap Z)$, we have $X \subset X' \subseteq Z$ and $X \subset X' \subseteq Y$. Then $\widetilde{sup}(X', T^k) = \sum_{i=\tau-k+1}^{\tau-1} \widetilde{sup}(Z, t_i) + \widetilde{sup}(Y, t_\tau) = \widetilde{sup}(X, T^k)$, implying that $X \sqsubset^{W_C} X'$ and hence $X \notin C$, which leads to a contradiction. Thus, $X = (Y \cap Z)$.
(*If*) Since $\widetilde{sup}(X,T^k) \geq minsup(k)$ for some $k \in \{2, \ldots, w\}$, it remains to prove that $\forall X' \supset X$, $\widetilde{sup}(X, T^k) > \widetilde{sup}(X', T^k)$. Suppose to the contrary that $\exists X' \supset X$ such that $\widetilde{sup}(X', T^k) = \widetilde{sup}(X, T^k)$. Then, $\widetilde{sup}(X', t_\tau) = \widetilde{sup}(X, t_\tau)$ and $\sum_{i=\tau-k+1}^{\tau-1} \widetilde{sup}(X', t_i) = \sum_{i=\tau-k+1}^{\tau-1} \widetilde{sup}(X, t_i)$. Since $X \sqsubset^{t_\tau} Y$, $X \sqsubset^{W_L} Z$, we have $X \subset X' \subseteq Y$ and $X \subset X' \subseteq Z$. Thus, $X' \subseteq (Y \cap Z) = X$, which contradicts $X \subset X'$. Thus, $X \in C$ follows. □

Example 3 Let $\sigma = 0.001$, $r = 0.5$, $t_\tau = t_5$, $W_L = \langle t_1, \ldots, t_4 \rangle$ and $W_C = \langle t_2, \ldots, t_5 \rangle$. We assume a uniform input rate of 5,000 transactions over each time unit t_i. We compute $minsup(k)$ to be (18, 12, 7, 3) for $k = (4, 3, 2, 1)$ over both W_L and W_C. Table 3 shows eight itemsets and indicates which of the sets, L, F and C, the itemsets belong to. It also records the approximate support of the itemsets over each t_i.

By Theorem 1(a), all semi-FCIs in F are in C. Thus, `b`, `bd`, `abc` and `abcd` are in C.

Both `bc` and `abd` are in L but not in F. By Theorem 1(b), `bc` is included in C since $\widetilde{sup}(\texttt{bc}, T^4) = 25 \geq minsup(4) = 18$, $\texttt{bc} \sqsubset^{t_5} \texttt{abc}$, and $\widetilde{sup}(\texttt{bc}, T^4) > \widetilde{sup}(\texttt{abc}, T^4)$. However, `abd` is not in C since $\widetilde{sup}(\texttt{abd}, T^4) = (5 + 6 + 3 + 3) = 17 < minsup(4) = 18$, while $\widetilde{sup}(\texttt{abd}, T^3) = \widetilde{sup}(\texttt{abcd}, T^3) = 12 \geq minsup(3) = 12$, which implies that $\texttt{abd} \sqsubset^{W_C} \texttt{abcd}$ (or more precisely, $\texttt{abd} \sqsubset^{T^3} \texttt{abcd}$, since `abcd` is a 3-semi-FCI in C).

Table 3 Semi-FCIs in L, F and C, respectively

	L	F	C	t_1	t_2	t_3	t_4	t_5
b	√	√	√	9	11	9	8	6
g	√			7	5	3	3	0
ab			√	8	5	6	3	5
bc	√		√	7	7	8	5	5
bd	√	√	√	9	7	7	6	4
abc		√	√	7	3	6	3	5
abd	√			8	5	6	3	3
abcd	√	√	√	7	3	6	3	3

Finally, among all the subsets of the semi-FCIs in F that are not in L, only ab satisfies the constraints in Theorem 1(c); that is, ab $\sqsubset^{t_5}$ abc, ab $\sqsubset^{W_L}$ abd, abc $\notin L$, (abc $\cap$ abd) = ab, and $\widetilde{sup}(\texttt{ab}, T^4) = 19 \geq minsup(4) = 18$.

Theorem 1(a) states that we must process every semi-FCI in F, while Theorems 1(b) and 1(c) show that we do not need to enumerate exhaustively all subsets of a semi-FCI in F. By Theorem 1(b), all subsets that are in L need to be updated. However, most subsets of a semi-FCI in F are not semi-FCIs over W_L and hence these subsets are not in L. Fortunately, Theorem 1(c) imposes constraints that a subset $X \notin L$ of some $Y \in F$ needs to be processed only if (1) $Y \notin L$; (2) $\exists Z \in L$ such that $X \sqsubset^{W_L} Z$; and (3) $Y \cap Z = X$. In practice, only a small portion of the subsets satisfy all these constraints, and the following theorem further reduces the number of such subsets to be processed.

Theorem 2 *Given X and Y, where $X \sqsubset^{t_\tau} Y$, $X \notin L$ and $Y \notin L$. If $\exists Z \in L$ such that $X \subset Z \subset Y$, then $X \notin C$.*

Proof Since $X \notin L$, $Z \in L$ and $X \subset Z$, there must exist $X' \in L$, where $X \subset X' \subseteq Z$, such that $X \sqsubset^{W_L} X'$. Since $X \subset X' \subseteq Z \subset Y$, we have $X' \sqsubset^{t_\tau} Y$. Thus, it follows that $X \sqsubset^{W_C} X'$ and hence $X \notin C$. □

It follows from Theorem 2 that, after processing a subset Z of $Y \in F \backslash L$, if $Z \in L$, we can skip processing all $X \subset Z$ where $X \notin L$.

Example 4 In Table 3, after we process bc as a subset of abc, since bc is in L, we can skip processing all subsets of bc that are not in L; thus, the subset c is not processed.

5.4 Incremental update algorithm

We incorporate the results of all the lemmas and theorems into our algorithm, which we call *IncMine*, as shown in Algorithm 3. From Theorem 1(a), all semi-FCIs in F are in C and, hence, we add them to L (Line 11), which is returned as C at the end of the update (Line 29). For any subset X of $Y \in F$, where $X \notin L$, Theorem 1(c) states that X is in C iff Y is not in L. Therefore, when Y is in L, we only process its subsets that are in L (Lines 2–9), while we process those subsets that are not in L only when Y is not in L (Lines 11–23). When a subset of Y has been updated previously, by Lemma 2, we skip processing all its subsets (Lines 6–7, 17–18). By Theorem 2, after

Algorithm 3 *IncMine*(F,L)

```
1. for each Y ∈ F in size-ascending order do
2.     if (Y ∈ L)
3.         ComputeK(Y, 1);
4.         for each X ⊂ Y in size-descending order do
5.             if (X ∈ L)
6.                 if (X is updated)
7.                     skip processing X and all its subsets;
8.                 else
9.                     UpdateSubsetInL(X, Y);
10.    else /* Y ∉ L */
11.        L ← L ∪ {Y};
12.        if (∃Z ∈ L such that Z ⊐^{W_L} Y)
13.            sup~(Y, t_i) ← sup~(Z, t_i), for τ − w + 1 ≤ i ≤ τ − 1;
14.        ComputeK(Y, 1);
15.        for each X ⊂ Y in size-descending order do
16.            if (X ∈ L)
17.                if (X is updated)
18.                    skip processing X and all its subsets;
19.                else
20.                    UpdateSubsetInL(X, Y);
21.                    skip processing all subsets of X that are not in L;
22.            else /* X ∉ L */
23.                UpdateSubsetNotInL(X, Y);
24. for each X ∈ L in size-descending order do
25.    if (X is not updated)
26.        k = ComputeK(X, 1);
27.        if (k > 0 and ∃Z ∈ L such that Z ⊐^{W_C} X)
28.            L ← L − {X};
29. return C ← L;
```

we process a subset X that is in L, we skip processing all subsets of X that are not in L (Line 21) and only process X's subsets that are in L, as we do in Lines 2–9.

When we update a subset X of a semi-FCI $Y \in F$, we assign Y's support over t_τ to X (Line 1 of Procedure 1 and Line 3 of Procedure 2). When we update an itemset that is not in L, we obtain its support over $\langle t_{\tau-w+1}, \ldots, t_{\tau-1}\rangle$ from its SFS in L (Lines 12–13 of Algorithm 3 and Lines 1–2 of Procedure 2).

Procedures 1 and 2 are direct implementations of Theorems 1(b) and 1(c), respectively, while Procedure 3 computes the value of k such that X is a k-semi-FCI. Procedure 3 also deletes the unpromising support records of X or completely deletes X if it is not a semi-FCI.

Procedure 1 *UpdateSubsetInL*(X,Y)

```
1. sup~(X, t_τ) ← sup~(Y, t_τ);
2. k = ComputeK(X, 2);
3. if (k > 0 and ∃X' ∈ L such that X ⊂ X' ⊆ Y and sup~(X', T^k) = sup~(X, T^k))
4.     L ← L − {X};
```

Procedure 2 *UpdateSubsetNotInL*(X,Y)

1. **if** $(\exists Z \in L$ such that $Z \sqsupset^{W_L} X$ and $Y \cap Z = X)$
2. $\quad \widetilde{sup}(X,t_i) \leftarrow \widetilde{sup}(Z,t_i)$, for $\tau - w + 1 \leq i \leq \tau - 1$;
3. $\quad \widetilde{sup}(X,t_\tau) \leftarrow \widetilde{sup}(Y,t_\tau)$;
4. $\quad k = ComputeK(X, 2)$;
5. $\quad$ **if** $(k > 0)$
6. $\quad\quad L \leftarrow L \cup \{X\}$;

Procedure 3 *ComputeK*(X, k')

1. $K \leftarrow \{k : (k' \leq k \leq w) \wedge (\widetilde{sup}(X, T^k) \geq minsup(k))\}$;
2. **if** $(K \neq \emptyset)$
3. $\quad k \leftarrow MAX(K)$;
4. $\quad$ delete $\widetilde{sup}(X, t_i), \forall i < \tau - k + 1$;
5. $\quad$ **return** k;
6. **else**
7. $\quad$ delete X;
8. $\quad$ **return** 0;

Finally, Lines 24-28 of Algorithm 3 update those itemsets in L that are infrequent in t_τ and delete them if they are no longer semi-FCIs over W_C. In Line 27, since X is in L, we need to search for X's SFS, Z, in L. If there is no such Z in L, then X is in C.

We also give a comprehensive example of how L is updated with F to give C. However, we delay the example until Section 6.2 where we can illustrate the update process together with the inverted index structure that supports the efficient processing of the algorithm.

5.5 Operations in the incremental update

In this subsection, we identify the major operations in Algorithm 3 and Procedures 1–3, as listed below, and then discuss how they are processed.

OP1: Selection of a semi-FCI from L (Lines 2,5 and 16 of Algorithm 3).

OP2: Selection of the SFS of an itemset from L (Lines 12 and 27 of Algorithm 3, and Line 1 of Procedure 2). (Note that we do not perform the selection of SFS in Line 3 of Procedure 1 but simply pass $\widetilde{sup}(X', T^k)$, where X' is the most recently processed superset of X, into the procedure call.)

OP3: Insertion (Line 11 of Algorithm 3 and Line 6 of Procedure 2) of a semi-FCI into L.

OP4: Deletion (Line 28 of Algorithm 3, Line 4 of Procedure 1 and Line 2 of Procedure 3) of a semi-FCI from L.

OP5: Subset enumeration of an itemset, with skip of repeated subsets (Lines 4, 7, 15, 18 and 21 of Algorithm 3).

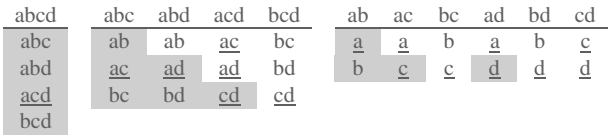


Fig. 2 Subset enumeration of abcd

The efficient processing of OPs 1 to 4 depends on the data structures used to store the sets F, L and C, which we discuss in Section 6. In the remainder of this section, we discuss the processing of OP5, subset enumeration with skip of repeated subsets.

To skip processing all subsets of an updated itemset, we enumerate the subsets of each semi-FCI in F in descending order of their sizes and in lexicographic order for subsets of the same size. Given two subsets, X and Y, of an itemset, a large portion of the subsets of X and Y may overlap. If X has been updated, we want to avoid Y's subsets that are also the subsets of X being enumerated, so that these overlapping subsets will not be processed repeatedly. We illustrate subset enumeration by the following example.

Example 5 Consider enumerating the subsets of the itemset abcd, as shown in Fig. 2. There are three iterations, which enumerate subsets of sizes 3, 2 and 1, respectively. A recursive procedure is invoked to enumerate only the shaded subsets shown in the figure in order to avoid duplication.

Suppose that, in the first iteration, we find that the subset acd has been updated and hence we want to skip processing the subsets of acd, i.e., the underlined itemsets shown in the figure. Although the recursive algorithm will only enumerate the shaded subset, cd, of acd, the other two unshaded subsets, ac and ad, will be enumerated as subsets of abc and abd, respectively. But by noticing that the second unshaded column, i.e., ac and ad, corresponds to the second shaded row, we can pass this information into the recursive procedure to skip enumerating ac and ad. Thus, only ab, bc and bd are enumerated in the second iteration. In the last iteration, we only need to enumerate the subsets of ab since we have skipped ac and ad. Similarly, since the unshaded subset, a, of ac corresponds to the shaded subset, a, of ab, we skip processing a and only enumerate b as the subset of ab.

Due to space constraints, we do not discuss the recursive procedure in detail here but remark that for itemsets of greater size, we need to skip their overlapping subsets across multiple unshaded columns and shaded rows. However, the underlying idea is similar to the case of the single unshaded column and shaded row shown in Example 5.

6 Data structure

In this section, we describe a data structure used to store F, L and C in order to support the efficient processing of the update operations in Algorithm 3. Since the semi-FCIs in F are ordered according to their size and processed one by one, we simply use an array to keep F. Thus, we only discuss the data structure for storing L. Note that C is just L at the end of the update at each window slide.

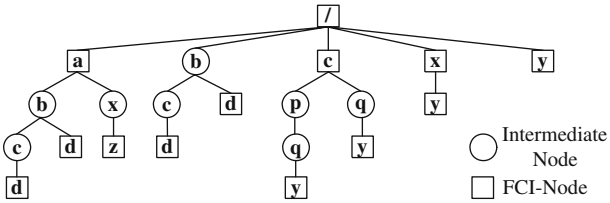


Fig. 3 A prefix-tree structure

6.1 The prefix-tree structure

We first consider storing L in a prefix tree structure, which is a data structure prevalently used in mining FIs or FCIs (Han et al. 2000; Zaki and Hsiao 2002; Wang et al. 2003; Giannella et al. 2004; Chi et al. 2004). However, using a prefix tree to store a set of FCIs may not necessarily save space because the subsets of an FCI may not be FCIs. For example, in Fig. 3, only the squared nodes represent FCIs while the circled nodes are intermediate nodes.

A disadvantage of using the prefix tree is that the search space for finding the SFS of an itemset is very large, since a superset may contain some items that are ordered before all the items in the subset. For example, to select the SFS of `py`, that is, `cpqy` in Fig. 3, we need to traverse (either by depth-first or breadth-first) the first two *irrelevant* subtrees that are rooted at the nodes "`a`" and "`b`," respectively.

Considering that the selection of the SFS is a crucial operation in the incremental update, we propose an inverted index for the efficient processing of the operation.

6.2 The inverted index structure

We first partition L according to the size of the semi-FCIs in L such that all semi-FCIs of the same size belong to the same partition. Each partition is stored in an array, called the *FCI-array*, and each semi-FCI in the FCI-array is assigned an *ID*, which is taken as the position of the semi-FCI in the array. This ID is then used to look up the semi-FCI in the inverted index. We call an FCI-array the *size-n FCI-array* if the semi-FCIs in the array are of size n. The approximate support of a semi-FCI computed over each time unit is also kept with the semi-FCI in the FCI-array. We also associate a queue, called the *garbage-queue*, with each size-n FCI-array. When a semi-FCI is deleted from an FCI-array, we push the ID (position) of the semi-FCI into the garbage-queue. When a semi-FCI is to be inserted into an FCI-array, we store it in the position (ID) popped out of the garbage-queue. If the queue is empty, then we attach the semi-FCI to the end of the array.

Inverted FCI Index We now define the *Inverted FCI Index* (*IFI*). The IFI has the following components:

- An array, called the *Item Array* (*IA*), stores the set of all items, $\mathcal{I}$, in lexicographic order.
- Each item in the IA is associated with a list of variable-length arrays called *ID-arrays*. Each ID-array in the list stores a set of IDs in ascending order of their integral values. The IDs in each ID-array belong to semi-FCIs of the same size.

Table 4 Four FCI-arrays

ID	Size-1	Size-2	Size-3	Size-4
0	x	xy	xyz	bxyz
1	y	xz	bxy	abcd
2	b	bx	bxz	
3	g	by	abd	
4		bc		
5		bd		

Thus, an ID-array that stores IDs of size-n semi-FCIs is called a *size-n ID-array* in the list.

– Given a semi-FCI, $X = \{x_1, \ldots, x_n\}$, in L, we store its ID in the size-n ID-array of each item x_i in the IA, for $1 \leq i \leq n$.

Example 6 Table 4 shows the size-n FCI-arrays, where $n = 1, 2, 3, 4$. The ID of each semi-FCI represents its position in the corresponding array. Figure 4 shows the corresponding IFI. For example, the ID of the semi-FCI bxz is "2"; thus, we have "2" in the size-3 ID-arrays of the items b, x and z in the IA.

IFI-based Incremental Update To illustrate the incremental update using the IFI, we consider the semi-FCIs shown in Table 3 and update L with F to give C. We add a few dummy semi-FCIs (i.e., those semi-FCIs that consist of the items x, y or z) to L to give a more intuitive view of the ID-arrays. These dummy semi-FCIs, together with the semi-FCIs in Table 3, are shown in Table 4. However, we do not consider updating the dummy semi-FCIs.

The first semi-FCI in F to be processed is b. We locate b in the IA and immediately obtain its ID "2" in the size-1 ID-array. Thus, we access Position 2 of the size-1 FCI-array to update the support of b.

We then process the next semi-FCI bd in F. We locate the items b and d in the IA and join the size-2 ID-arrays of b and d to obtain the ID of bd, "5." We access Position 5 of the size-2 FCI-array to update the support of bd. Then, we continue to process the subsets of bd. We do not process b since it has been updated previously. When we enumerate to d, since its ID is not in the size-1 ID-array of d, we know that d is not in L. Since bd is in L, by Theorem 1(c), d is not in C and thus not processed.

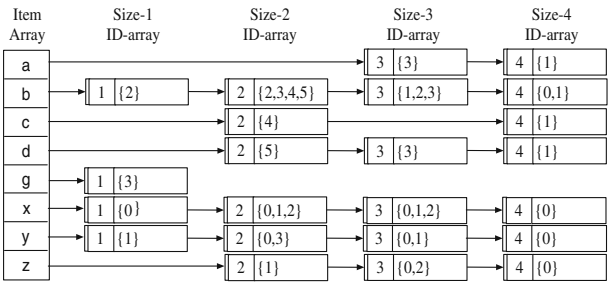


Fig. 4 An inverted FCI index (IFI)

Next, we process abc. Since the join on the size-3 ID-arrays of the items a, b and c produces no result, abc is not in L. Thus, we need to find the SFS of abc in L. We continue the join with the size-4 ID-arrays of a, b and c and obtain the ID "1." Then, we access Position 1 of the size-4 FCI-array and obtain the support of abc over W_L from its SFS abcd. We store abc in Position 4 of the size-3 FCI-array and insert its ID "4" into the size-3 ID-arrays of the items a, b and c. Note that although the ID of abc is now in the ID-arrays, our algorithm does not consider a newly updated semi-FCI as the SFS of any itemset.

Then, we process the subsets of abc (by Lines 15–23 of Algorithm 3). We first process ab. Since we cannot find the ID of ab, we want to find its SFS in L. We join the size-3 ID-array of a and b and obtain "3," which is the ID of abd. Since the intersection of abc and abd is ab, we compute the support of ab and find that ab is a semi-FCI. We therefore add ab to the size-2 FCI-array and insert its ID into the size-2 ID-arrays of a and b. For ac, since it is not in L, we find its SFS, which is abcd. Since abcd $\cap$ abc = abc $\neq$ ac, by Theorem 1(c), ac is not in C and is thus not processed. We also update bc, which is in L, with $\widetilde{sup}$(abc,t_τ), and skip processing its subset c by Theorem 2. For a, we do not process it due to the same reason as ac. We skip processing b since it has been updated previously.

Lastly, we process abcd. Since abcd is in L, we process Lines 3–9 of Algorithm 3. We update abcd similarly to how we update bd. Then, we process its first subset, abc. Since abc has been updated, by Lemma 2, we skip processing abc and all its subsets. We then process abd and find that, as explained in Example 3, abd $\sqsubset^{W_C}$ abcd. Thus, we delete abd from the size-3 FCI-array, push its ID, "3," into the size-3 garbage-queue, and then remove "3" from the size-3 ID-arrays of the items a, b and d. By Lines 3–9 of Algorithm 3, acd, bcd, ad and cd are not processed since they are not in L. Other subsets of abcd are the subsets of the updated semi-FCIs abc and bd and thus are skipped.

Finally, we scan the FCI-arrays once to update those semi-FCIs that are infrequent over t_τ. Thus, g is deleted since it no longer satisfies the semi-FCI requirement.

We describe the IFI-based operations in Operations 1 to 4. Operations 1 selects an itemset from L, while Operation 2 selects the SFS of an itemset from L. Operations 3 and 4 insert and delete an itemset and its ID into/from the FCI-array and the ID-arrays, respectively.

Operation 1 *Select*($X = \{x_1, \ldots, x_n\}$)

1. Locate x_i, for $1 \leq i \leq n$, in the IA;
2. Perform join on the size-n ID-arrays of all x_i;
3. Return the join result, if any, as the ID of X;

Operation 2 *SelectSFS*($X = \{x_1, \ldots, x_n\}$)

1. $j = 1$;
2. Join the size-$(n+j)$ ID-arrays of all x_i, for $1 \leq i \leq n$;
3. Increment j until the join obtains a result for some size-$(n+j)$ ID-arrays or terminate the join when the end of the list of ID-arrays of some x_i is reached;
4. Return the join result, if any, as the ID of X's SFS;

Operation 3 *Insert*($X = \{x_1, \ldots, x_n\}$)

1. **if** (the size-n garbage-queue is empty)
2. Store X at the end of the size-n FCI-array;
3. **else** /* Some semi-FCIs in the size-n FCI-array were deleted. */
4. ID $\leftarrow$ POP(size-n garbage-queue);
5. Store X in Position ID of the size-n FCI-array;
6. Insert ID in the size-n ID-arrays of all x_i, for $1 \leq i \leq n$;

Operation 4 *Delete*($X = \{x_1, \ldots, x_n\}$)

1. Push X's ID, i.e., its position in the size-n FCI-array, into the size-n garbage-queue;
2. Delete X from the size-n FCI-array;
3. Delete X's ID from the size-n ID-arrays of all x_i, for $1 \leq i \leq n$;

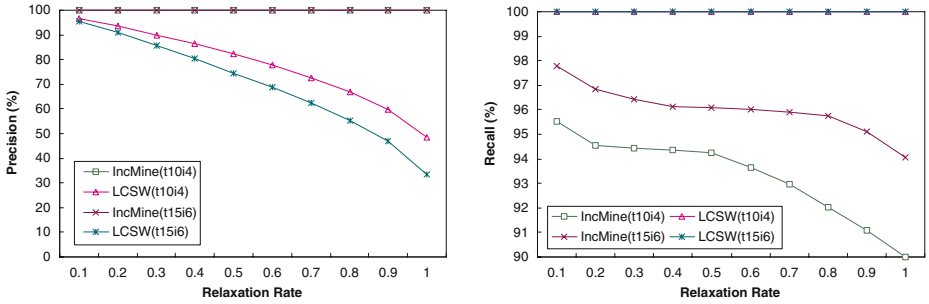
6.3 Efficiency of IFI-based operations

Although a semi-FCI of size n has its ID duplicated n times in the ID-arrays, the memory consumption of the IFI is not large since the set of semi-FCIs is small in most cases, as verified by our experimental results. However, the advantage of using the ID-array is that joining ordered arrays is simple and extremely fast. Since the IDs in each ID-array are distinct, the number of comparisons in the worst case is the total number of IDs in the ID-arrays. In most cases the join terminates early since it produces at most one ID. Although insertion and deletion of IDs are more costly, they are not often performed.

7 Performance evaluation

IncMine uses CHARM (Zaki and Hsiao 2002) to compute the set of semi-FCIs over each time unit. We implement IncMine in C++ and evaluate its performance on a Sun Ultra-SPARC III with 900 MHz CPU and 4GB RAM. We compare IncMine's performance with that of Moment (Chi et al. 2004), which represents the state-of-the-art streaming algorithm for mining FCIs. Since Moment is an exact algorithm, we also compare IncMine's performance with that of a variant of the *Lossy Counting* algorithm (Manku and Motwani 2002), which mines FIs over a sliding window. We denote this variant of Lossy Counting as *LCSW* in our experiments. We remark that LCSW, which updates a batch of incoming or expiring transactions at each window slide, is different from the algorithm proposed by Chang and Lee (2004), which updates on each incoming or expiring transaction. We implement both algorithms and find that the algorithm by Chang and Lee is much slower than LCSW and runs out of our 4 GB of memory in most cases.

Datasets We generate the data streams using the IBM data generator (Agrawal and Srikant 1994; IBM Quest 1996). However, we find that the datasets are too sparse, leading to the result that in many cases almost all FIs are FCIs. Although



a Precision **b** Recall

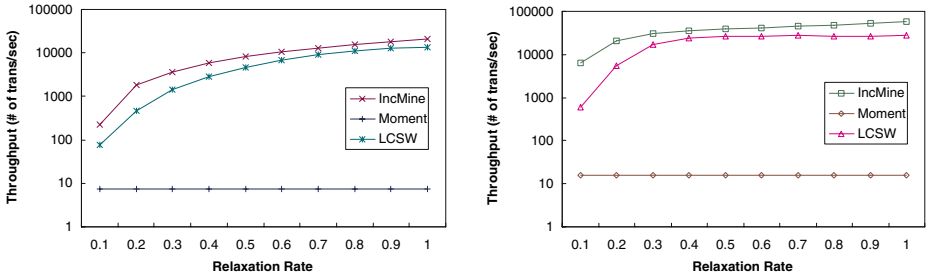
Fig. 5 Precision and recall with varying relaxation rate

existing work on mining FCIs (Zaki 2000; Zaki and Hsiao 2002; Wang et al. 2003) has verified that many real datasets (FIMI Dataset Repository 2003) are dense and hence the number of FCIs can be orders of magnitude smaller than that of FIs, most real datasets are not large enough to model a stream. To show the effectiveness of our streaming FCI mining algorithm, we modify the IBM data generator in order to generate denser data streams.

The original generator first produces a set of patterns, each of which is associated with a probability. To generate a transaction, patterns are picked up according to their associated probabilities to constitute the transaction until the required length of that transaction is reached. The generated dataset is sparse because the patterns are lowly correlated and the chance that two patterns are always chosen to generate the same transaction is very low due to the large number of patterns.

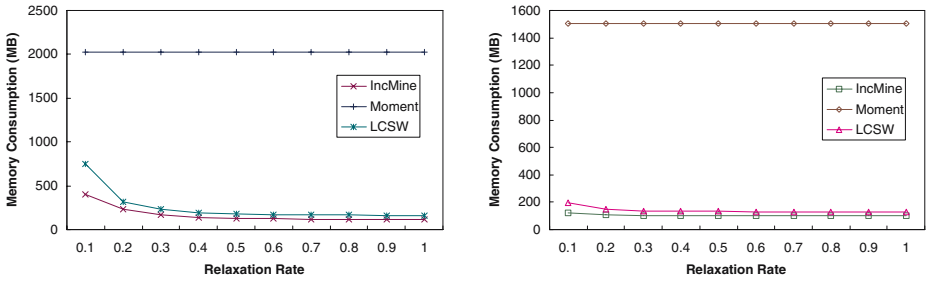
We modify the generator as follows. We produce a set of disjoint patterns, each of which is also associated with a probability. For each pattern, we further produce a small set of sub-patterns, which are subsets of the pattern. To generate a transaction, we first choose a pattern according to its probability. Then, we randomly either use the pattern or pick up one of its sub-patterns to construct the transaction. Thus, the probability that the subsets of a pattern occur in different transactions is increased.

We generate two types of data streams, t15i6 and t10i4, where 15 and 10 (6 and 4) are the average size of a transaction (a maximal FI) of the two streams, respectively.



a Stream t15i6 **b** Stream t10i4

Fig. 6 Throughput with varying relaxation rate



a Stream t15i6 **b** Stream t10i4

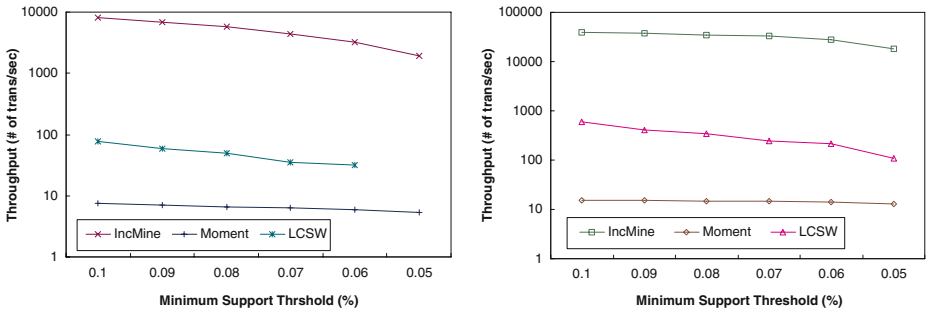
Fig. 7 Memory consumption with varying relaxation rate

The number of unique items over each stream is 10,000. Each stream consists of 3×10^6 transactions and we report the results averaged over all the window slides. Each window consists of 20 time units and each time unit receives approximately 50,000 transactions. We control the ratio of the number of FCIs to that of FIs over each window to be within [0.1, 0.2]. We remark that the smaller the ratio, the better is the performance of IncMine than that of LCSW.

7.1 Varying relaxation rate

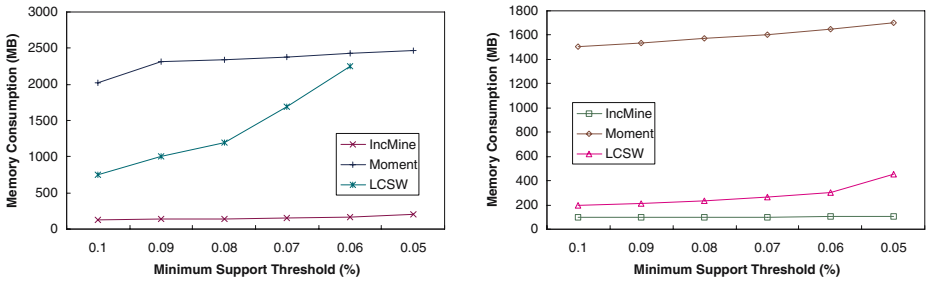
We first examine the effect of varying the relaxation rate r on IncMine and LCSW (note that $\epsilon = r\sigma$ in LCSW). We set $\sigma = 0.1\%$ and vary r from 0.1 to 1. We assess the *precision* and the *recall* of the mining results obtained by the different algorithms. Here the precision and the recall are respectively defined as $(|A \cap B|/|B|)$ and $(|A \cap B|/|A|)$, where A and B are the actual set and the approximate set of FIs over each window, respectively. The approximate set of FIs is either returned by LCSW or recovered from the FCIs obtained by IncMine.

We report the precision and recall of IncMine and LCSW in Figs. 5a and b, for both streams t15i6 and t10i4. The precision and recall of Moment are omitted from the figures since Moment is an exact algorithm. The figures show that IncMine has



a t15i6 **b** t10i4

Fig. 8 Throughput with varying minimum support threshold

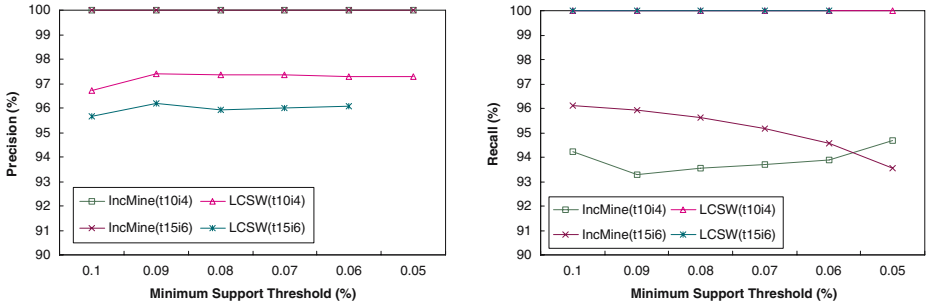


a t15i6 **b** t10i4

Fig. 9 Memory consumption with varying minimum support threshold

100% precision while LCSW has 100% recall. Although Fig. 5a shows that LCSW attains high precision when r is small, its precision drops linearly to be less than 40% with the increase in r (i.e., the increase in the error parameter $\epsilon = r\sigma$). On the contrary, when r increases from 0.1 to 1, the recall of IncMine drops only slightly from 98 to 94% for t15i6 and from 95.5 to 90% for t10i4. The experimental results reveal that the estimation mechanism of Lossy Counting relies on ϵ to control the accuracy, while our progressively increasing *minsup* function maintains a high accuracy which is only slightly affected by the change in r.

We also report the *throughput* (in logarithmic scale) and the memory consumption of the three algorithms in Figs. 6 and 7. Here throughput is measured as the number of transactions processed per second by the algorithms. Figures 6 and 7 show that both IncMine and LCSW attain significantly higher throughput and consumes less memory when r becomes larger. This result is reasonable because the two algorithms mine over each time unit using a relaxed MST $r\sigma$ and as a result, increasing r gives rise to faster mining process and less memory consumption. It is a very important finding as evidenced in Figs. 5a and b that IncMine maintains a high accuracy even for large r. As a result, we can use a larger r to achieve much faster speed and less memory consumption but with only slightly degraded accuracy, as we show in the next section.



a Precision **b** Recall

Fig. 10 Precision and recall with varying minimum support threshold

7.2 Varying minimum support threshold

In this experiment, we assess the performance of IncMine on different MST σ. We vary σ from 0.1 to 0.05%. According to Lossy Counting (Manku and Motwani 2002), a good choice of ϵ is 0.1σ. Thus, we set $r = 0.1$ for LCSW. However, the experimental results in the previous section shows that IncMine can obtain a good accuracy even with a larger r. Thus, we set $r = 0.5$ for IncMine. We note that LCSW runs out of memory when $\sigma = 0.05\%$ on the data stream t15i6 (lower values of σ are thus not tested).

Figures 8a and b report the throughput, in logarithmic scale, of the three algorithms. The result verifies the capability of IncMine to handle high-speed streams as it can process up to 10,000 and 40,000 transactions per second for t15i6 and t10i4, respectively. For all σ tested and for both streams, the throughput of IncMine is over two orders of magnitude higher than that of LCSW and three orders of magnitude higher than that of Moment.

Figures 9a and b show that IncMine achieves a roughly constant memory consumption of no more than 200 MB. In all cases, the memory consumption of IncMine is considerably less than that of LCSW and substantially less than that of Moment.

Figures 10a and b verify that IncMine achieves 100% precision and high recall of over 93% in all cases. Although LCSW also achieves high precision and 100% recall, Figs. 8 and 9 show that IncMine runs significantly faster and consumes much less memory than does LCSW.

8 Concluding remarks

In this paper, we study the problem of incrementally maintaining an approximate set of FCIs over a sliding window. We introduce the notion of semi-FCIs, which enables us to progressively increase the minimum support threshold for an itemset as it is retained longer in the window, thereby drastically reducing the number of itemsets that need to be kept and processed. We observe that the majority of the subsets of a semi-FCI will not be semi-FCIs and need not be updated. This leads to the design of an efficient algorithm, IncMine, for the incremental update. In addition, we also develop a useful inverted index structure to facilitate the update operations.

We demonstrate in our experiments that IncMine significantly outperforms both the Moment and Lossy Counting algorithms, in both throughput and memory consumption, and that IncMine achieves highly accurate approximation results. Compared with the exact algorithm Moment, IncMine is able to handle very high-speed streams at the cost of only slightly lowered recall. Such approximate but high-quality online answers are particularly well-suited to the exploratory nature of most practical stream mining applications, such as trend analysis and fraud/anomaly detection, where the main goal is to identify generic, interesting or unexpected patterns rather than provide results that are exact to the last decimal. Compared to Lossy Counting, IncMine is much less sensitive to the error parameter and is able to achieve significantly higher throughput by increasing the error parameter, while still attains high accuracy.

We note that the progressively increasing function, *minsup*, can be characterized for specific streams such as one having unevenly distributed data, and our proposed

IncMine is adaptable for any specific *minsup* functions. Our on-going work is to further improve the efficiency of IncMine by developing a model to generate specific *minsup* functions according to the characteristics of specific data streams and to tune the functions to accommodate with the changes in a stream over time.

Acknowledgements We thank Mr. Yun Chi and Prof. Mohammed J. Zaki for providing us with the Moment and the Charm source codes, respectively.

References

Agrawal, R., Imielinski, T., & Swami, A. N. (1993). Mining association rules between sets of items in large databases. In *SIGMOD*, (pp. 207–216).
Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *VLDB*, (pp. 487–499).
Chang, J. H., & Lee, W. S. (2003). Finding recent frequent itemsets adaptively over online data streams. In *KDD*, (pp. 487–492).
Chang, J. H., & Lee, W. S. (2004). A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineering, 20*(4), 753–762.
Chen, Y., Dong, G., Han, J., Wah, B. W., & Wang, J. (2002). Multi-dimensional regression analysis of time-series data streams. In *VLDB*, (pp. 323–334).
Chi, Y., Wang, H., Yu, P. S., & Muntz, R. R. (2004). Moment: Maintaining closed frequent itemsets over a stream sliding window. In *ICDM*, (pp. 59–66).
FIMI Dataset Repository (2003). Frequent itemset mining dataset repository. http://fimi.cs.helsinki.fi/data/.
Garofalakis, M. N., Gehrke, J., & Rastogi, R. (2002). Querying and mining data streams: You only get one look a tutorial. In *SIGMOD*, (p. 63).
Giannella, C., Han, J., Pei, J., Yan, X., & Yu, P. S. (2004). *Mining frequent patterns in data streams at multiple time granularities*. Cambridge, MA: MIT Press.
Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. In *SIGMOD*, (pp. 1–12).
IBM Quest (1996). Ibm quest data mining project. frequent itemset mining dataset repository. http://www.almaden.ibm.com/software/quest/.
Jiang, N. & Gruenwald, L. (2006). CFI-Stream: Mining closed frequent itemsets in data streams. In *KDD*, (pp. 592–597).
Kifer, D., Ben-David, S., & Gehrke, J. (2004). Detecting change in data streams. In *VLDB*, (pp. 180–191).
Lee, C.-H., Lin, C.-R., & Chen, M.-S. (2001). Sliding-window filtering: An efficient algorithm for incremental mining. In *CIKM*, (pp. 263–270).
Li, H., Lee, S., & Shan, M. (2004). Algorithm for mining frequent itemsets over the entire history of data streams. In *Proc. of First International Workshop on Knowledge Discovery in Data Streams*.
Manku, G. S. & Motwani, R. (2002). Approximate frequency counts over data streams. In *VLDB*, pages 346–357.
Pasquier, N., Bastide, Y., Taouil, R., & Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. In *ICDT*, (pp. 398–416).
Pei, J., Han, J., & Mao, R. (2000). CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, (pp. 21–30).
Wang, J., Han, J., & Pei, J. (2003). CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *KDD*, (pp. 236–245).
Yu, J. X., Chong, Z., Lu, H., & Zhou, A. (2004). False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *VLDB*, (pp. 204–215).
Zaki, M. J. (2000). Generating non-redundant association rules. In *KDD*, (pp. 34–43).
Zaki, M. J., & Hsiao, C.-J. (2002). Charm: An efficient algorithm for closed itemset mining. In *SDM*.