Mining Frequent Itemsets in Time-Varying Data Streams

Abstract

A transactional data stream is an unbounded sequence of transactions continuously generated, usually at a high rate. Mining frequent itemsets in such a data stream is beneficial to many real-world applications but is also a challenging task since data streams are unbounded and have high arrival rates. Moreover, the distribution of data streams can change over time, which makes the task of maintaining frequent itemsets even harder. In this paper, we propose a false-negative oriented algorithm, called TWIM, that can find most of the frequent itemsets, detect distribution changes, and update the mining results accordingly. TWIM uses two tumbling windows, one for maintenance and one for change prediction. We maintain a frequent itemset list and a candidate list for a data stream. Every time the two windows tumble, we check members in both lists. New frequent itemsets will be added and itemsets no longer frequent will be removed. Experimental results show that our algorithm performs as good as other false-negative algorithms on data streams without *distribution change, and has the ability to detect changes over time-varying data streams in real-time with a high accuracy rate.*

1. Introduction

The problem of mining frequent itemsets has long been recognized as an important issue for many applications such as fraud detection, trend learning, customer management, marketing and advertising. Mining frequent itemsets in data stream applications is also beneficial for a number of purposes such as knowledge discovery, trend learning, fraud detection, transaction prediction and estimation [9, 12, 19]. However, the characteristics of stream data – unbounded, continuous, fast arriving, and time-changing – make this a challenging task. Existing mining techniques that focus on relational data cannot handle streaming data well [10].

First, since a data stream is unbounded and usually has high arrival rate, it is not possible to rescan the whole stream, and thus, multi-scan data mining algorithms for traditional databases and batch data cannot be applied to stream data directly. Second, the large volume of the data makes it infeasible to process the entire stream within limited memory [2]. Finally, fast data streams are created by continuous activities over long periods of time, usually months or years. It is natural that the underlying processes generating them can change over time, and thus, the data distribution may show important changes during this period. This is referred to as *data evolution*, *time-varying data*, or *concept-drifting data* [13, 17, 24]. Updating and maintaining frequent itemsets for such time-varying data streams in real time is a challenging issue.

The problem of mining frequent items has been extensively studied [5, 8, 9, 15]. The common assumptions that the total number of items is too large for memory-intensive solutions to be feasible. Mining frequent items over a data stream under this assumption still remains an open problem. However, the task of mining frequent itemsets is much harder than mining frequent items. Even when the number of distinct items is small, the number of itemsets could still be exponential in the number of items, and maintaining frequent itemsets requires considerably more memory.

Mining frequent itemsets is a continuous process that runs throughout a data stream's life-span. Since the total number of itemsets is exponential, it is impractical to keep statistics for each itemset due to bounded memory. Therefore, usually only the itemsets that are already known to be frequent are recorded and monitored, and statistics of other infrequent itemsets are discarded. However, as mentioned, data streams can change over time. Hence, an itemset that was once infrequent can become frequent if a stream changes its distribution. Detecting such changes is an important task especially for online applications, such as leak detection, network monitoring, and decision support. However, since it is not feasible to maintain all itemsets, it is hard to detect frequent itemsets when distribution changes happen. Furthermore, even if we could detect these itemsets, we would not be able to obtain their statistics (supports), since mining a data stream is a one-pass procedure and history information is not retrievable. Distribution changes over data streams might have considerable impact on the mining results, but few of the previous works have addressed this issue.

A number of techniques have been proposed in recent years for mining frequent itemsets over streaming data. However, as we discuss in Section 3, they have problems in meeting common requirements: ability to process large number of itemsets in real time, low (preferably minimum) memory usage, and ability to cope with time varying data streams.

In this paper, we develop a new algorithm, called TWIM, that can find most of the frequent itemsets in real time. It can also predict the distribution change and update the mining results accordingly. Our approach maintains two tumbling windows over a data stream: a maintenance window and a prediction window.

All current frequent itemsets are recorded and maintained in the maintenance window, and we use the prediction window to keep track of candidates that have the potential of becoming frequent if the distribution of stream values changes. Every time the windows tumble, we check if new frequent itemsets and candidates should be added, and if some existing ones need to be removed from the lists. Since we do not keep statistics for every itemset within the windows, memory usage is limited. Experimental results show that TWIM is as effective as previous approaches for *non-time-varying* data streams, but is superior to them since it can also capture the distribution change for time-varying streams in real-time.

The contributions of this paper can be summarized as follows:

- We formalize the problem of mining for frequent itemsets over streams, and prove that the problem of mining the complete frequent itemsets over data streams is NP-hard.
- We introduce a double-tumbling windows model for mining frequent itemsets over data streams. Experimental results show that this model can adapt to distribution changes effectively and efficiently. Since one of the windows is a virtual window, memory usage for this model is not larger than that of the popular sliding window models.
- We present a novel algorithm, TWIM, to mine frequent itemsets over data streams. Unlike most of the existing frequent itemset mining techniques, our algorithm is false-negative oriented and is suitable for streams with distribution changes. Furthermore, since TWIM maintains counters for candidate itemsets long before they become frequent, once an itemset becomes frequent, its estimated support is more accurate than those algorithms that can only detect and record changes in the last minute.

The rest of the paper is organized as follows. In Section 2, we formally define our problem and prove that mining the complete frequent itemsets is NP-hard. In Section 3, we discuss the related work and compare them with our proposal. In Section 4, the proposed TWIM algorithm for detecting frequent itemsets over time-varying data streams is presented. Experimental results are given in Section 5. We conclude the paper in Section 6.

2. Problem Statement

Let $\mathcal{I} = \{i_1, i_2, ..., i_n\}$ be a set of *items*. A *transaction* T accesses a subset of items $I \subseteq \mathcal{I}$. A data stream is an unbounded sequence of tuples that continuously arrive in real time. In this paper, we are interested in transactional data streams, where each tuple corresponds to a transaction. Example of such transaction-based data streams include online commerce, web analysis, banking, and telecommunications applications, where each transaction accesses a set of items from a certain item pool, such as inventory, customer list, or a list of phone numbers.

Let $\mathcal{T}_t = \{T_1, T_2, ..., T_{N_t}\}$ be the set of transactions at time t. N_t is the total number of transactions received up to time t. The data stream that contains $\mathcal{T}_t$ is denoted by $D_{\mathcal{T}_t}$. Note that the number of items, n, is finite and usually not very large, while the number of transactions, N_t, will grow monotonically as time progresses.

Table 1 summarizes the main symbols used in this paper.

Definition 1. Given a transaction $T_j \in \mathcal{T}_t$, and a subset of items $\mathcal{A} \subseteq \mathcal{I}$, if T_j accesses $\mathcal{A}$ (i.e., $\mathcal{A} \subseteq I_j$), then we say T_j *supports* $\mathcal{A}$.

Table 1. Meanings of symbols used

Symbols	Meanings
t, t'	timestamps we always let $t' > t$ in this paper
T_j	a transaction
I_j	an itemset that T_j accesses
$D_{\mathcal{T}_t}$	transactional data stream
N_t	number of transactions at time t
$\mathcal{I}$	set of items
$\mathcal{A}$	an itemset
$sup(\mathcal{A})$	$\mathcal{A}$'s counter number of transactions that support $\mathcal{A}$
$S(\mathcal{A})$	the support of $\mathcal{A}$ at time t $S(\mathcal{A}) = sup(\mathcal{A})/N_t$
$\mathcal{A}_{\mathcal{T}_t}$	set of frequent itemsets
$\mathcal{C}$	set of candidates
δ	minimum support for frequent itemsets if $S(\mathcal{A}) \geq \delta$ then $\mathcal{A} \in \mathcal{A}_{\mathcal{T}_t}$
θ	minimum support for candidates if $\theta \leq S(\mathcal{A}) < \delta$ then $\mathcal{A} \in \mathcal{C}$
W_M	maintenance window
W_P	prediction window

Definition 2. Let $sup(\mathcal{A})$ be the total number of transactions that support $\mathcal{A}$. If $S(\mathcal{A}) = sup(\mathcal{A})/N_t > \delta$, where δ is a predefined threshold value, then $\mathcal{A}$ is a frequent itemset in $D_{\mathcal{T}_t}$ under current distribution. $S(\mathcal{A})$ is called the *support* of $\mathcal{A}$.

Example 1. Consider a data stream $D_{\mathcal{T}_t}$ with $\mathcal{T}_t = \{T_1, T_2, T_3, T_4, T_5\}$ at time t and a set of items $\mathcal{I} = \{a, b, c, d\}$. Let $I_1 = \{a, b, c\}$, $I_2 = \{a, b, c, d\}$, $I_3 = \{c, d\}$, $I_4 = \{a\}$, and $I_5 = \{a, c, d\}$. If threshold $\delta = 0.5$, then the frequent itemsets are $\mathcal{A}_1 = \{a\}, \mathcal{A}_2 = \{c\}, \mathcal{A}_3 = \{d\}, \mathcal{A}_4 = \{a, c\}$, and $\mathcal{A}_5 = \{c, d\}$, with supports $S(\mathcal{A}_1) = S(\mathcal{A}_2) = 0.8$, and $S(\mathcal{A}_3) = S(\mathcal{A}_4) = S(\mathcal{A}_5) = 0.6$.

Let $\mathcal{A}_{\mathcal{T}_t} = \{\mathcal{A}_1, \mathcal{A}_2, ...\mathcal{A}_m\}$ be the complete set of frequent itemsets in $D_{\mathcal{T}_t}$ under current distribution. The ultimate goal of mining frequent itemsets in data stream $D_{\mathcal{T}_t}$ is to find $\mathcal{A}_{\mathcal{T}_t}$ in polynomial time with limited memory space. However, it has been proven that the problem of finding $\mathcal{A}_{\mathcal{T}_t}$ off-line is NP-hard [21]. The following theorem proves that on-line updating $\mathcal{A}_{\mathcal{T}_t}$ for a data stream that grows in real-time is also NP-hard.

THEOREM 1. *The problem of finding the complete set of frequent itemsets $\mathcal{A}_{\mathcal{T}_t}$ in a given transaction-based data stream $D_{\mathcal{T}_t}$ with threshold δ is NP-hard.*

PROOF. If there exists an algorithm that can list all frequent itemsets $\mathcal{A}_{\mathcal{T}_t}$ in polynomial time, then this algorithm should also be able to count the total number of such frequent itemsets with the same efficiency. Thus, it suffices to show that counting $|\mathcal{A}_{\mathcal{T}_t}|$ for any given $D_{\mathcal{T}_t}$ and threshold δ is NP-hard.

Let n be the total number of items in $\mathcal{I}$, and N_t be the total number of transactions at time t. Construct a $n \times N_t$ matrix $\mathcal{M}$. Each element $M_{k,j}$ in $\mathcal{M}$ is a Boolean value: $M_{i,j} = 1$ iff $i_k \in T_j$, and 0 otherwise. Hence, there exists a one-to-one mapping between stream $D_{\mathcal{T}_t}$ and matrix $\mathcal{M}$.

Any $n \times N_t$ matrix $\mathcal{M}$ can be mapped to a monotone-2CNF formula with N_t clauses and n variables. Therefore, we can reduce the problem of counting $|\mathcal{A}_{\mathcal{T}_t}|$ to the problem of counting the number of satisfying assignments for a monotone-2CNF formula using polynomial time.

It has been proven that the problem of counting the number of satisfying assignment of monotone-2CNF formulas with threshold δ is #P-hard [11, 23]. Hence, counting $|\mathcal{A}_{\mathcal{T}_t}|$ is a NP-hard problem. □

Note that in the proof, N_t does not have to be infinite, and does not even have to be a large number. Therefore, even if techniques such as windowing that can reduce the number of transactions N_t are applied, the problem of mining the complete set of frequent itemsets still remains NP-hard. Furthermore, the size of the complete set of frequent itemsets $\mathcal{A}_{\mathcal{T}_t}$ can be exponential. An extreme case is that every transaction T_j in $D_{\mathcal{T}_t}$ accesses $\mathcal{I}$ (i.e. $\forall T_j \in \mathcal{T}_t$, $I_j = \mathcal{I}$). For such cases, no algorithm can list $\mathcal{A}_{\mathcal{T}_t}$ using polynomial time and space. However, note that this proof holds even for the cases where $|\mathcal{A}_{\mathcal{T}_t}|$ is not exponential. Even when the actual size of $\mathcal{A}_{\mathcal{T}_t}$ is small, the time taken for *searching for* $\mathcal{A}_{\mathcal{T}_t}$ is still exponential.

3. Related Works

Mining frequent items and itemsets is a challenging task and has attracted attention in recent years. Jiang and Gruenwald [14] provide a good review of research issues in frequent itemsets and association rule mining over data streams.

The problem of mining frequent *items* and approximating frequency counts has been extensively studied [5, 8, 9, 15]. Many of the works consider mainly the applications where total number of items in a stream is very large, and therefore, under memory-intensive environments, it is not possible to store a counter even for each of the items. However, the problem of mining frequent items is much easier than the problem of mining frequent itemsets. Even when the number of distinct items is small, which is true for many applications, the number of itemsets could be exponential.

One of the classical frequent itemset mining techniques for relational DBMSs is Apriori [1], which is based on the heuristic that if one itemset is frequent, then its supersets may also be frequent. Apriori requires multiple scans over the entire data and hence cannot be directly applied in a streaming environment. Many Apriori-like approaches for mining frequent itemsets over streaming data have been proposed in literature [4, 6, 16], and some of them can be applied on dynamic data streams. However, as will be discussed in Section 4.2, Apriori-based approaches suffer from a long delay when discovering large sized frequent itemsets, and may miss some frequent itemsets that can be easily detected using TWIM.

Yang and Sanver [25] propose a naive approach that can only mine frequent itemsets and association rules that contain only few items (usually less than 3). When the sizes of potential frequent itemsets are over 3, this algorithm may take intolerably long time to execute.

Manku and Motwani propose the Lossy Counting (LC) algorithm for mining frequent itemsets [18]. LC prunes itemsets with low frequency quickly, and thus only frequent itemsets will remain. Because LC has a very low runtime complexity and is easy to implement, it is one of the most popular stream mining techniques adopted in real-world applications. However, as experimentally demonstrated by a number of studies, LC may not perform well in practice [6, 8, 26], and is not applicable to data streams that change over time.

Chang and Lee [3] propose an algorithm named estDec for finding recent frequent itemsets by setting a decay factor. It is based on the insight that historical data should play a less important role in frequency counting. This approach does not have the ability to detect any itemsets that change from infrequent to frequent due to distribution drifts.

Chi et. al present an algorithm called Moment [7], which maintains *closed* frequent itemsets [22] using a tree structure named CET. The Moment algorithm provides accurate results within the window, and can update the mining results when stream distribution changes. However, Moment is not suitable for streams that change distributions frequently, because there might be a long overhead for updating CET when new nodes are added or an itemset is deleted. Furthermore, if the total number of the frequent itemsets or their size is large, Moment could consume a large amount of memory to store the tree structure and hash tables. Chang and Lee also adopt a sliding window model to mine recently frequent itemsets [4], which suffers from the same problem of memory usage boundary and may not be feasible in practice.

Most of the techniques proposed in literature are false-positive oriented, that is, the itemsets they find may not be truly frequent ones. False-positive techniques may consume more memory, and are not suitable for many applications where accurate results, even if not complete, are preferred. Yu et al propose a false-negative oriented algorithm, FDPM, for mining frequent itemsets [26]. The number of counters used in FDPM is fixed, and thus memory usage is limited. However, this approach cannot detect distribution changes in the stream, because an itemset could be pruned long before it becomes frequent.

4. TWIM: Algorithm for Mining Time-Varying Data Streams

In this section, we propose an algorithm called TWIM that uses two tumbling windows to detect and maintain frequent itemsets for any data stream. The algorithm is false-negative oriented: all itemsets that it finds are guaranteed to be frequent under current distribution, but there may be some frequent itemsets that it will miss. However, TWIM usually achieves high recall according to our experimental results. Since it is a false-negative algorithm, its precision is always 100%.

In order to detect distribution changes in time, we apply tumbling windows model on $D_{\mathcal{T}_t}$ (Section 4.1). A tumbling window accepts streaming transactions in "batches" that span a fixed time interval [20]. When windows tumble, the supports of existing frequent itemsets will be updated. If a distribution change occurs during the time span of the window, then some frequent itemsets may become infrequent, and vice versa. In most of the previous techniques, itemsets that are not frequent at the point when the check is performed are simply discarded. Since only the supports for frequent itemsets are maintained, the infrequent itemsets that become frequent due to the distribution change are hard to detect. Even if such itemsets can be somehow detected, since the historical information are not retrievable, their estimated supports may be far from the true values, which leads to poor precision. Therefore, we maintain a candidate list that contains a list of itemsets that have the potential to become frequent when the distribution of $D_{\mathcal{T}_t}$ changes. Since the supports for the candidates are maintained long before they become frequent, their estimated supports have high accuracy. How to predict candidate itemsets and the procedure for reducing the size of candidate lists in order to reduce memory usage are discussed in Section 4.2.

When windows tumble, the supports of all candidates are updated. If a distribution change occurs, some infrequent itemsets are added to and some itemsets will be removed from the candidate list according to certain criteria (Section 4.3). Candidates with supports greater than δ are moved to frequent itemset list.

The main TWIM algorithm is given in Algorithm 1. We expand each procedure in the following subsections. The experimental results show that TWIM is sensitive to distribution changes, and

can update its mining results accordingly in real-time.

Algorithm 1 TWIM Algorithm

```
1: INPUT: Transactional data stream D_{T_t}
2:        Tumbling window W_M and W_P
3:        Threshold δ and θ
4: OUTPUT: A list of frequent itemsets A_{T_t} and their supports
5: A_{T_t} = Φ; C = Φ; N_t = 0;
6: sup(i_1) = sup(i_2) = ... = sup(i_n) = 0;
7: for all transaction T_k that arrives in D_{T_t} do
8:   if W_M is not ready to tumble then
9:     Update the supports for all frequent itemsets and candidates
10:  else
11:    //Windows ready to tumble
12:    Call MAINTAIN_CURRENT_FREQSETS;
13:    //Move infrequent itemsets from A_{T_t} to candidates
14:    Call DETECT_NEW_FREQSETS;
15:    //Check if any itemset in candidate becomes frequent
16:    Call MAINTAIN_CANDIDATES;
17:    //Add new candidates
18:    Call UPDATE_CANDIDATE_SUP;
19:    //update supports for all candidates
20:    W_M and W_P tumble;
21:  end if
22: end for
```

4.1 Tumbling windows design

For most real-life data streams, especially the ones with distribution changes, recent data are more important than historical data. Based on this insight, we adopt a tumbling windows model to concentrate on recently arrived data.

We define a time-based tumbling window W_M for a given data stream, which we call the *maintenance window* since it is used to maintain existing frequent itemsets. Smaller W_M is more sensitive to distribution changes in $D_{\mathcal{T}_t}$, however, it will also incur higher overhead as the interval for updating frequent itemset list and candidate list is shorter. On the other hand, larger W_M reduces the maintenance overhead, but it cannot detect sudden distribution changes.

Since data streams are time-varying, a frequent itemset can become infrequent in the future, and vice versa. It is easy to deal with the first case. Since we keep counters for all frequent itemsets, we can check their supports periodically (every time W_M tumbles), and remove the counters of those itemsets that are no longer frequent. However, in the latter case, since we do not keep any information about the currently infrequent itemsets, it is hard to tell when the status changes. Furthermore, even if we can detect a new frequent itemset, we would not be able to estimate its support, as no history exists for it.

To deal with this problem, we define a second tumbling window called the *prediction window* (W_P) on the data stream. W_P moves together with W_M, aligning the window endpoints. It keeps history information for candidate itemsets that have the potential to become frequent. The size of W_P is larger than W_M, and it is predefined based on system resources, the threshold δ, and the accuracy requirement of the support computation for candidates. Note that we do not actually maintain W_P; it is a virtual window that is only used to keep statistics. Hence, the size (time length) of W_P can be as large as required. A large prediction window can ensure high accuracy of the estimated supports for candidate itemsets, resulting in high precision. However, it cannot detect sudden distribution changes, and may consume more memory as there are more itemsets maintained in the window. A smaller W_P is more sensitive to distribution changes and requires less memory, but the precision of the mining result may be lower.

Figure 1 demonstrates the relationship between maintenance window W_M and prediction window W_P with an example. In Figure 1, W_M and W_P are the windows before tumbling, while W'_M and W'_P are windows afterwards[1]. When the end of W_M is reached, it will tumble to the new position W'_M. Every time W_M tumbles, W_P will tumble at the same time. This is to ensure that the endpoints of W_M and W_P are always aligned, so that frequent itemsets and candidate itemsets can be updated at the same time. Therefore, in Figure 1, W_P tumbles to its new position W'_P even before its time interval is fully spanned.

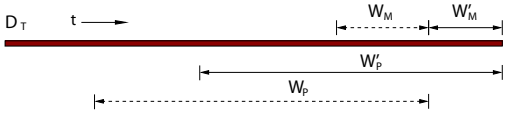

Figure 1. Tumbling windows for a data stream

Mining frequent itemsets requires keeping counters for all itemsets; however, the number of itemsets is exponential. Consequently, it is not feasible to keep a counter for all of them, and thus, we only keep counters for the following:

- A counter for each item $i_j \in I$. Since total number of items n is small (typically less than tens of thousands), it is feasible to keep a counter for each item. If each counter is 4 bytes, then the memory requirement for storing all the counters usually will not exceed 4 MB.
- A counter for each identified frequent itemset. As long as the threshold value δ is reasonable (i.e., not too low), the number of frequent itemsets will not be large.
- A counter for each itemset that has the potential to become frequent. We call these *candidate itemsets*[2]. The list of all candidates is denoted as $\mathcal{C}$. The number of candidate itemsets $|\mathcal{C}|$ is also quite limited, as long as the threshold value θ (discussed in Section 4.2) is reasonable.

If a frequent itemset becomes infrequent at some point, instead of deleting it right away, we move it from the set of frequent itemsets $\mathcal{A}_{\mathcal{T}_t}$ to $\mathcal{C}$, and reset its counter (but not remove it just in case it becomes frequent again soon, as will be explained in more detail in Section 4.3). Hence, the counter for an itemset is removed only when this itemset is removed from candidate list $\mathcal{C}$.

4.2 Predicting candidates

To deal with the difficulties of determining which infrequent itemsets may become frequent, as discussed in the previous sections, we introduce a prediction stage to generate a list of candidate itemsets $\mathcal{C}$, which includes itemsets that are most likely to become frequent. The prediction stage happens as W_M and W_P tumble, so that statistics for these candidates can be collected within the new window W'_P.

Any itemset $\mathcal{A}$ with $\theta \leq S(\mathcal{A}) < \delta$ is considered a candidate and included in $\mathcal{C}$. Here θ is the support threshold for considering an itemset as a candidate. Every time W_P tumbles, we evaluate all candidates in $\mathcal{C}$. If the counter of one candidate itemset is below θ, it is removed from $\mathcal{C}$ and its counter is released. θ is user defined: smaller θ may result in a higher recall, but consumes more

[1] In this paper, if we need to discuss the maintenance and prediction windows before and after tumbling, we always use W_M and W_P to denote the old windows before tumbling, and W'_M, and W'_P to denote the new windows after tumbling.

[2] The question of which itemsets are predicted to have such potential will be discussed in the following subsections.

memory since more candidates are generated; a high θ value can reduce memory usage by sacrificing the number of resulting frequent itemsets. Thus, the θ value can be set based on application requirements and available memory.

Every time W_M and W_P tumble, the counters of all candidates and the supports of all items will be updated. If one candidate itemset $\mathcal{A}' \in \mathcal{C}$ becomes frequent, then $\forall \mathcal{A}'' \in \mathcal{A}_{\mathcal{T}_t}$, $\mathcal{A} = \mathcal{A}' \cup \mathcal{A}''$ might be a candidate. Similarly, if one infrequent item i becomes frequent at the time windows tumble, then $\forall \mathcal{A}'' \in \mathcal{A}_{\mathcal{T}_t}$, $\mathcal{A} = \{i\} \cup \mathcal{A}'$ can be a candidate.

One simple solution is to add all such supersets $\mathcal{A}$ into the candidate list $\mathcal{C}$. However, this will result in a large increase of the candidate list's size, since the total number of $\mathcal{A}$ for each $\mathcal{A}'$ or $\{i\}$ can be $|\mathcal{A}_{\mathcal{T}_t}|$ in the worst case. The larger the candidate list, the more memory required for storing counters, and the longer it takes to update the list when W_M and W_P tumble.

As indicated earlier, many existing frequent itemset mining techniques for streams are derived from the popular Apriori algorithm [1]. When an itemset $\mathcal{A}'$ with size k is determined to be frequent, Apriori makes multiple passes to search for its supersets. In the first run (or in our streaming case, the first time W_M and W_P tumble after $\mathcal{A}'$ is detected), all its supersets with size $k+1$ are added to the candidate list. The size of candidate supersets increases by 1 at every run, until the largest itemset is detected. This strategy successfully reduces the number of candidates; however, in cases if the itemset size $|\mathcal{I}|$ is large, it may take extremely long time until one large frequent itemset is detected.

Example 2. Let $\mathcal{I} = \{a, b, c, d, e\}$, where $\{a\}, \{b\}, \{c\}$ and $\{d\}$ are frequent itemsets. Assume that, at the point when W_M and W_P tumble, item e becomes frequent, hence, $\{e\}$'s immediate supersets $\{a, e\}, \{b, e\}, \{c, e\}$ and $\{d, e\}$ will be regarded as candidates. If, by next time W_M and W_P tumble, $\{a, e\}$ is detected as frequent, then, $\{a, b, e\}, \{a, c, e\}$ and $\{a, d, e\}$ will be added to the candidate list. Assuming the largest itemset $\{a, b, c, d, e\}$ is actually a frequent itemset, it will take time $4 \times |W_M|$ for this itemset to be detected. When the maintenance window size is large, this delay could be unacceptably long. Furthermore, if the distribution of the stream changes rapidly, the itemset $\{a, b, c, d, e\}$ may never be detected as frequent.

Another problem may occur for such Apriori-like approaches, as demonstrated in the following example.

Example 3. Let $\mathcal{I} = \{a, b, c, d\}$, where a and b are frequent items, and itemset $\{a, b\}$ is the only candidate. Assume that next time windows tumble, $S(\{a, b\}) < \theta$, and hence, itemset $\{a, b\}$ will be discarded from the candidate list $\mathcal{C}$. Assuming t time later, c becomes a frequent item, we will have $\mathcal{A}_{\mathcal{T}_t} = \{\{a\}, \{b\}, \{c\}\}$, and $\mathcal{C} = \{\{a, c\}, \{b, c\}\}$. If by the next run, both $\{a, c\}$ and $\{b, c\}$ are determined to be frequent, then we might end up with $\mathcal{A}_{\mathcal{T}_t} = \{\{a\}, \{b\}, \{c\}\, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Notice the problem here: itemset $\{a, b\}$ is not included in $\mathcal{A}_{\mathcal{T}_t}$. However, since $\{a, b, c\}$ is a frequent itemset, by definition, $\{a, b\}$ must be frequent as well. The problem occurs because $\{a, b\}$ has been discarded long before. When the distribution changes and $\{a, b\}$ turns from infrequent to frequent, it cannot be added to the candidate list if $\{a\}$ and $\{b\}$ are in $\mathcal{A}_{\mathcal{T}_t}$ all the time. Although by simply adding all subsets of $\{a, b, c\}$ in $\mathcal{A}_{\mathcal{T}_t}$ we will be able to add $\{a, b\}$ back to the frequent itemset list, since Apriori-like approaches only check the supersets of the existing frequent itemsets, the subsets of existing frequent itemsets are not considered.

Continuing with Example 3, assume that item d becomes frequent at time t'. Using Apriori-like approaches, it will take $3 \times |W_M|$ to detect the frequent itemset $\{a, b, c, d\}$. However, if instead, we start from the current largest itemset in $\mathcal{A}_{\mathcal{T}_t}$, that is $\{a, b, c\}$ in this example, then itemset $\{a, b, c, d\}$ is considered a candidate, and can be detected as frequent next time windows tumble. Hence, the time for detecting $\{a, b, c, d\}$ is only $|W_M|$. By definition, the complete $\mathcal{A}_{\mathcal{T}_t}$ can be obtained by simply computing the power set of $\{a, b, c, d\}$ minus null set ϕ. This approach minimize both the delay in detection and the size of candidate list.

Definition 3. Given an itemset list $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_m\}$, for $\forall \mathcal{A}' = \{\mathcal{A}'_1, \mathcal{A}'_2, ...\mathcal{A}'_r\}$, where $\mathcal{A}'_1, \mathcal{A}'_2, ...\mathcal{A}'_r \in \mathcal{A}$, if $\mathcal{A}'_1 \cup \mathcal{A}'_2 \cup ... \cup \mathcal{A}'_r = \mathcal{A}_1 \cup \mathcal{A}_2 \cup ... \cup \mathcal{A}_m$ and $r < m$, then we say $\mathcal{A}'$ is a *cover set* of $\mathcal{A}$, denoted as $\mathcal{A}^C$.

For example, given itemset list $\mathcal{A} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, b, c\}\}$,$\mathcal{A}^C = \{\{d\}, \{a, b\}, \{a, b, c\}\}$ is one cover set.

Definition 4. Given an itemset list $\mathcal{A}$ and all its cover set $\mathcal{A}^C_1, \mathcal{A}^C_2, ..., \mathcal{A}^C_q$, if $|\mathcal{A}^C_s| = min(\forall |\mathcal{A}^C_i|)$, where $i = 1, ..., q$, then we call $\mathcal{A}^C_s$ the *smallest cover set* of $\mathcal{A}$, denoted as $\mathcal{A}^{SC}$.

For example, the smallest cover set $\mathcal{A}^{SC}$ of itemset list $\mathcal{A} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, b, c\}\}$ is $\{\{d\}, \{a, b, c\}\}$.

When a candidate itemset or an infrequent item becomes frequent, the candidate list can be expanded from either direction, i.e., combining the new frequent itemset with all current frequent items in $\mathcal{A}_{\mathcal{T}_t}$ or with the smallest cover set of $\mathcal{A}_{\mathcal{T}_t}$. The decision as to which direction to follow depends on the application. If the sizes of the potential frequent itemsets are expected to be large, then the smallest cover set could be a better option. On the other hand, if small sized frequent itemsets are more likely, then Apriori-like approaches can be applied. However, in many real-world scenarios, it is hard to make such predictions, especially when the distribution of the data streams is changing over time. Hence, we apply a hybrid method in our approach.

4.2.1 Hybrid approach for generating candidates

Our hybrid candidate prediction technique is as follows. At the time W_M and W_P tumble:

- **Step 1.** Detect new frequent itemsets and move them from candidate set $\mathcal{C}$ into set of frequent itemsets $\mathcal{A}_{\mathcal{T}_t}$. Also detect any new frequent items and add them into $\mathcal{A}_{\mathcal{T}_t}$. This step will be discussed in detail in Section 4.3.
- **Step 2.** Update $\mathcal{A}_{\mathcal{T}_t} = \mathcal{A}_{\mathcal{T}_t} \cup \mathcal{P}(\mathcal{A}^{SC}_{\mathcal{T}_t}) - \phi$, where $\mathcal{P}(\mathcal{A}^{SC}_{\mathcal{T}_t})$ is power set of $\mathcal{A}_{\mathcal{T}_t}$'s smallest cover set. This step is for eliminating the problem discussed in Example 3.
- **Step 3.** Detect itemsets in $\mathcal{C}$ whose supports have fallen below θ. Replace each of these itemsets by its subsets of length one smaller, and then remove it from $\mathcal{C}$. For example, if itemset $\{a, b, c, d\}$ is not a candidate anymore, then we add itemsets $\{a, b, c\}, \{a, b, d\}, \{b, c, d\}$ and $\{a, c, d\}$ into $\mathcal{C}$, and then remove $\{a, b, c, d\}$. This process can be regarded as the reverse process of a Apriori-like approach.
- **Step 4.** Set $\mathcal{C} = \mathcal{C} - \mathcal{A}_{\mathcal{T}_t}$. After Steps 2 and 3, there could be some candidates that are already included in $\mathcal{A}_{\mathcal{T}_t}$, hence we do not need to keep them in the candidate list $\mathcal{C}$ anymore.
- **Step 5.** Let $\mathcal{A}'$ be one candidate itemset that becomes frequent, or $\{j\}$ where j is an item that turns from infrequent to frequent.

  Step 5.1. $\forall \mathcal{A} = \{i\} \cup \mathcal{A}'$, where $i \in (\mathcal{I} - \mathcal{A}')$ and $\{i\} \in \mathcal{A}_{\mathcal{T}_t}$, if $\mathcal{A}$ is not in $\mathcal{A}_{\mathcal{T}_t}$, then $\mathcal{A}$ is a new candidate.

  Step 5.2. $\forall \mathcal{A}'' \in (\mathcal{A}_{\mathcal{T}_t} - \mathcal{A}')^{SC}$, if $\mathcal{A} = \mathcal{A}'' \cup \mathcal{A}'$ is not in $\mathcal{A}_{\mathcal{T}_t}$, then $\mathcal{A}$ is a new candidate.

Example 4. Let $I = \{a, b, c, d\}$, $\mathcal{A}_{\mathcal{T}_t} = \{\{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$, and $\mathcal{C} = \phi$. At the time W_M and W_P tumble:

Step 1. Assume that item d becomes frequent, hence $\mathcal{A}_{\mathcal{T}_t} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$.

Step 2. $\mathcal{A}_{\mathcal{T}_t} = \mathcal{A}_{\mathcal{T}_t} \cup \mathcal{P}(\mathcal{A}^{SC}_{\mathcal{T}_t}) - \phi = \mathcal{A}_{\mathcal{T}_t} \cup \mathcal{P}(\{\{d\}, \{a, b, c\}\}) - \phi = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Notice that itemset $\{b, c\}$ is added to $\mathcal{A}_{\mathcal{T}_t}$.

Steps 3 and 4. Since currently $\mathcal{C} = \phi$, these two steps are skipped.

Step 5. $\mathcal{C} = \{\{a, d\}, \{b, d\}, \{c, d\}, \{a, b, c, d\}\}$.

After time $|W_M|$, the two windows tumble again:

Case 1: $sup(\{a, b, c, d\}) \geq \theta$ and $\{a, b, c, d\}$ becomes frequent.

Step 1.1. $\mathcal{A}_{\mathcal{T}_t} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}, \{a, b, c, d\}\}$.

Step 1.2. $\mathcal{A}_{\mathcal{T}_t} = \mathcal{P}(\{a, b, c, d\}) - \phi$.

Step 1.3. $\mathcal{C} = \{\{a, d\}, \{b, d\}, \{c, d\}\}$.

Step 1.4. $\mathcal{C} = \mathcal{C} - \mathcal{A}_{\mathcal{T}_t} = \phi$.

Step 1.5. All frequent itemsets detected.

Case 2: $S(\{a, b, c, d\}) < \theta$, and no new frequent itemset detected.

Step 2.1 and step 2.2. $\mathcal{A}_{\mathcal{T}_t}$ remains unchanged.

Step 2.3. $\mathcal{C} = \{\{a, d\}, \{b, d\}, \{c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}$.

Step 2.4. Itemset $\{a, b, c\}$ is removed from $\mathcal{C}$ because it is already a frequent itemset.

Step 2.5. No new frequent itemset detected, thus this step does not apply.

Property: For each itemset $\mathcal{A}$ with size k that moves from infrequent to frequent at tumbling point t, let $\mathcal{C}_\mathcal{A}$ be the list of new candidates generated using our hybrid approach at Step 5. Let $|\mathcal{C}_\mathcal{A}|$ be the number of itemsets in $\mathcal{C}_\mathcal{A}$, and β be the total time required for all frequent itemsets in $\mathcal{C}_\mathcal{A}$ to be detected. We can prove that $|\mathcal{C}_\mathcal{A}| + \frac{2}{|W_M|}\beta \leq 2p - k$, where p is the total number of frequent items in $\mathcal{A}_{\mathcal{T}_t}$. (The proof is omitted due to page limit.)

Notice that p, i.e. the number of frequent items, is determined by the nature of the stream and is not related to the chosen mining method. This property indicates that the time and memory usage of our hybrid candidate generation approach are correlated. They are bounded to a constant that is not related to the size of minimal cover set $\mathcal{A}^{SC}_{\mathcal{T}_t}$. If, at time t, the size of $\mathcal{C}_\mathcal{A}$ is large (which indicates a large amount of memory consumption), then from this property, we know that the time for detecting all frequent itemsets in $\mathcal{C}_\mathcal{A}$ will be very short, i.e., large $|\mathcal{C}_\mathcal{A}|$ value indicates a small β. Note that once all the frequent itemsets are detected, $\mathcal{C}_\mathcal{A}$ will be removed from $\mathcal{C}$, therefore, the large memory usage only lasts for a short time period. On the other hand, if it takes longer to detect all frequent itemsets in $\mathcal{C}_\mathcal{A}$, then the memory usage will be quite limited, i.e., when β is large, $|\mathcal{C}_\mathcal{A}|$ must be small. Hence, this nice property guarantees that the overall memory usage of the proposed hybrid approach is small, and its upper bound is only determined by the number of frequent items in the stream.

4.2.2 Finding smallest cover set

Our candidate prediction technique uses smallest cover set of $\mathcal{A}_{\mathcal{T}_t}$ to discover the most number of frequent itemsets in the shortest time. In this section, we present an approximate algorithm that can find a good cover set[3] for a given frequent itemset list $\mathcal{A}_{\mathcal{T}_t}$ efficiently in terms of both time and memory.

- **Step 1.** Let $\mathcal{A}^{SC}_{\mathcal{T}_t} = \phi$. Build a set of itemsets $\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_m\}$. Let $\mathcal{B}_i = \mathcal{A}_i$ for $\forall \mathcal{A}_i \in \mathcal{A}_{\mathcal{T}_t}$.
- **Step 2.** Select the largest itemset $\mathcal{B}_k \in \mathcal{B}$, i.e., $|\mathcal{B}_k| = max(|\mathcal{B}_i|), \mathcal{B}_i \in \mathcal{B}, i = 1, ..., m$. If there is a tie, then select the one with larger corresponding itemset in $\mathcal{A}_{\mathcal{T}_t}$. In other words, if $|\mathcal{B}_k| = |\mathcal{B}_r| = max(|\mathcal{B}_i|)$ and $|\mathcal{A}_k| > |\mathcal{A}_r|$, where $\mathcal{A}_k$ and $\mathcal{A}_r$ are the corresponding frequent itemsets of $\mathcal{B}_k$ and $\mathcal{B}_r$ according to step 1, then select itemset $\mathcal{B}_k$. If there is still a tie, then randomly select one of the largest itemsets. Set $\mathcal{A}^{SC}_{\mathcal{T}_t} = \mathcal{A}^{SC}_{\mathcal{T}_t} \cup \{\mathcal{A}_k\}$.
- **Step 3.** For $\forall \mathcal{B}_i \in \mathcal{B}, i = 1, ..., m$, set $\mathcal{B}_i = \mathcal{B}_i - \mathcal{B}_k$. Remove all empty sets from $\mathcal{B}$.
- **Step 4.** If $\mathcal{B} = \phi$, then stop. Else go to step 2.

Example 5. Let $\mathcal{A}_{\mathcal{T}_t} = \{\{a, b, c\}, \{a, c, d\}, \{a, d, e\}, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}\}$.

Step 1. $\mathcal{A}^{SC}_{\mathcal{T}_t} = \phi$, and $\mathcal{B} = \mathcal{A}_{\mathcal{T}_t}$.

Step 2. Select the largest itemset $\mathcal{B}_1 = \{a, b, c\} \in \mathcal{B}$. $\mathcal{A}^{SC}_{\mathcal{T}_t} = \{\mathcal{A}_1\} = \{\{a, b, c\}\}$.

Step 3. $\mathcal{B}_2 = \{a, c, d\} - \{a, b, c\} = \{d\}$; $\mathcal{B}_3 = \{a, d, e\} - \{a, b, c\} = \{d, e\}$. All the rest itemsets in $\mathcal{B}$ are empty. Hence, $\mathcal{B} = \{\mathcal{B}_2, \mathcal{B}_3\} = \{\{d\}, \{d, e\}\}$. Go to Step 2.

Step 2-2. Select the largest itemset $\mathcal{B}_3 = \{d, e\} \in \mathcal{B}$. $\mathcal{A}^{SC}_{\mathcal{T}_t} = \mathcal{A}^{SC}_{\mathcal{T}_t} \cup \{\mathcal{A}_3\} = \{\{a, b, c\}, \{a, d, e\}\}$.

Step 3-2. $\mathcal{B} = \phi$. Algorithm terminates. The final cover set of $\mathcal{A}_{\mathcal{T}_t}$ is $\{\{a, b, c\}, \{a, d, e\}\}$.

The run time of this algorithm in the worst case is $(|\mathcal{A}_{\mathcal{T}_t}| - n) \times |\mathcal{A}^{SC}_{\mathcal{T}_t}|$, where n is the total number of frequent items in the stream. Hence, this algorithm is very efficient in practice.

4.2.3 Updating candidate support

For any itemset that changes its status from frequent to infrequent, instead of discarding it immediately, we keep it in the candidate list $\mathcal{C}$ for a while, in case distribution drifts back quickly and it becomes frequent again.

Every time W_M and W_P tumble, $\mathcal{C}$ is updated: any itemset $\mathcal{A} \in \mathcal{C}$ with $S(\mathcal{A}) < \theta$ along with its counter is removed, and new qualified itemsets are added resulting in the creation of new counters for them.

For an itemset $\mathcal{A}$ that has been in the candidate list $\mathcal{C}$ for a long time, if it becomes frequent at time t_i, its support may not be greater than the threshold δ immediately, because the historical transactions (i.e., the transactions that arrive in the stream before t_i) dominate in calculating $S(\mathcal{A})$. Therefore, in order to detect new frequent itemsets in time, historical transactions need to be eliminated when updating $S(\mathcal{A})$ for every $\mathcal{A} \in \mathcal{C}$.

Every time W_P tumbles, some of the old transactions will expire from W_P. For any itemset $\mathcal{A}$ that remains in $\mathcal{C}$, $S(\mathcal{A})$ is updated to eliminate the effect of those historical transactions that are no longer in W_P.

Since W_M and W_P are time-based tumbling windows, they tumble every $|W_M|$ time units. At the time W_M and W_P tumble, the transactions that expire from W_P are those transactions that arrived within the oldest $|W_M|$ time span in W_P. Hence, we can keep a checkpoint every $|W_M|$ time intervals in W_P, denoted as $chk_1, chk_2, ..., chk_p$, where chk_1 is the oldest checkpoint, and $p = \lfloor |W_P|/|W_M| \rfloor$. For each $\mathcal{A} \in \mathcal{C}$, we record the number of transactions arriving between chk_{i-1} and chk_i that access $\mathcal{A}$, denoted as $sup_i(\mathcal{A})$. When W_M and W_P tumbles, transactions

[3]Informally, a good cover set is one with a small number of itemsets, and the size of each itemset in this cover set is as large as possible. For example, $\{\{a, b, c\}, \{b, c, d\}\}$ is better than $\{\{a, b, c\}, \{d\}\}$, because if $\{b, c, d\}$ is determined to be frequent, many subsets can be added into $\mathcal{A}_{\mathcal{T}_t}$.

before checkpoint chk_1 are expired from W_P. Hence, $sup(\mathcal{A})$ is updated as $sup(\mathcal{A}) = sup(\mathcal{A}) - sup_1(\mathcal{A})$. Note that after tumbling, a new checkpoint is added, and chk_2 becomes the oldest checkpoint.

The procedures for maintaining candidate list $\mathcal{C}$ and updating candidate counters are given in Algorithm 2 and Algorithm 3, respectively.

Algorithm 2 MAINTAIN_CANDIDATES

```
1: //Step 1 is included in Algorithm 7.
2: A_{T_t} = A_{T_t} ∪ P(A^{SC}) − φ;
3: for all A ∈ C do
4:    if S(A) < θ then
5:       for all i ∈ A do
6:          C = C ∪ ({A − {i}})
7:       end for
8:       C = C − {A}; remove sup(A); remove offset(A);
9:       //we will explain the concept of offset in Section 4.4.
10:   end if
11: end for
12: C = C − A_{T_t};
13: for all A' = detect_new_freqset() do
14:    for all {i} ∈ A_{T_t} and i ∉ A' do
15:       A = {i} ∪ A';
16:       if A ∉ A_{T_t} then
17:          C = C ∪ {A}; sup(A) = 0; offset(A) = N_t;
18:       end if
19:    end for
20:    for all A_sc ∈ A^{SC} do
21:       A = A_sc ∪ A';
22:       if A ∉ A_{T_t} then
23:          C = C ∪ {A}; sup(A) = 0; offset(A) = N_t;
24:       end if
25:    end for
26: end for
```

Algorithm 3 UPDATE_CANDIDATE_SUP

```
1: for all A ∈ C do
2:    sup(A) = sup(A) − sup(A)_1;
3:    offset(A) = N_t;
4: end for
5: for all i = 2 to p do
6:    chk_{i−1} = chk_i; //expire the oldest point chk_1
7: end for
8: Set all the records in chk_p to 0;
9: //every time W_M tumbles, a new checkpoint is added to W_P
```

4.3 Maintaining current frequent itemsets and detecting new ones

Every time W_M tumbles, we update support values for all the existing frequent itemsets. If the support of an itemset $\mathcal{A}$ drops below δ, then we move it from the set of frequent itemsets $\mathcal{A}_{\mathcal{T}_t}$ to the candidate list $\mathcal{C}$, as indicated earlier, and the counter used to record its frequency will be reset to zero, i.e. $sup(\mathcal{A}) = 0$. This is to ensure that, if the distribution change is not rapid, $\mathcal{A}$ may stay in the candidate list for some time, as its history record plays a dominant role in its support. By resetting its counter, we eliminate the effect of historical transactions and only focus on the most recent ones. This ensures that the decrease in its support can be detected sooner. During the time-span of W_M, $sup(\mathcal{A})$ will be updated as each new transaction arrives. If, at the time W_M and W_P tumble, $S(\mathcal{A}) < \theta$, then $\mathcal{A}$ will be removed from the candidate list.

New frequent itemsets will come from either the infrequent items or the candidate list. Since we keep counters for all items $i \in \mathcal{I}$, when an item becomes frequent, it is easy to detect and its support is accurate. However, for a newly selected frequent itemset $\mathcal{A}$ that comes from candidate list $\mathcal{C}$, its support will not be accurate, as most of its historical information is not available. If we keep calculating its support as $S(\mathcal{A}) = sup(\mathcal{A})/N_t$, where N_t is the number of *all* transactions received so far, this $S(\mathcal{A})$ will not reflect $\mathcal{A}$'s true support. Hence, we need to keep an offset for $\mathcal{A}$, denoted $offset(\mathcal{A})$, that represents the number of transactions that were missed in counting the frequency of $\mathcal{A}$. $\mathcal{A}$'s support at any time $t' > t$ should be modified to $S(\mathcal{A}) = sup(\mathcal{A})/(N_{t'} - offset(\mathcal{A}))$, where $N_{t'}$ is the total number of transactions received at time t', as the data stream monotonically grows. Since the counters of candidate itemsets are updated every time W_M and W_P tumble to eliminate the history effect (as mentioned in Section 4.2), their offsets also need to be reset to the beginning of the new W_P.

Figure 2 demonstrates how the offset is calculated. Assume that an itemset $\mathcal{A}$ is added to the candidate list at the beginning of W_P (time t), and a counter is created for it. At the time W_M and W_P tumble (time t'), we need to calculate $S(\mathcal{A})$ to see if we can move $\mathcal{A}$ to the set of frequent itemsets $\mathcal{A}_{\mathcal{T}_{t'}}$. Since we do not have $\mathcal{A}$'s historical information before time t, we need to adjust $\mathcal{A}$'s offset to N_t. Hence, we know that $offset(\mathcal{A}) = N_t$, where t is the timestamp when $\mathcal{A}$ starts being recorded.

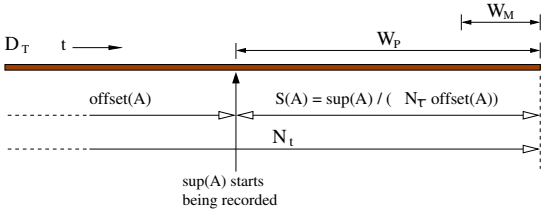

Figure 2. Offset for itemset $\mathcal{A}$

Note that the supports for such itemsets are no longer based on the whole history, unlike all items that we track throughout the entire life-span of the stream. However, using supports that only depend on recent history should not affect TWIM's effectiveness. This is because the data stream is continuous with a distribution that changes over time, and hence, the mining results over such data stream is temporary – the result at time t_1 may not be consistent with the result at time t_2 ($t_1 < t_2$). Therefore, calculating supports using the whole history may not reflect the **current** distribution correctly or promptly, not to mention the huge amount of memory required for tracking the entire history for each itemset. Our experiments demonstrate that our approach is sensitive to both steady and slow changes, and rapid and significant changes, while the existing techniques cannot perform well, especially for the latter case.

5. Experiments

In this section, we present a series of experiments to evaluate TWIM's performance in comparison with three others: SW method [4], which is a sliding window based technique suitable for dynamic data streams, FDPM [26], which is also a false-negative algorithm, and Lossy Counting (LC) [18], which is a widely-adopted false positive algorithm. Since neither FDPM nor LC has the ability to detect distribution changes, we conduct the experiments in two stages. In the first stage, we compare these algorithms over data streams without distribution drift. In the second stage, we introduce time-varying data streams.

5.1 Experimental setup and data sets

Our experiments are carried out on a PC with 3GHz Pentium 4 processor and 1GB of RAM, running Windows XP. All algorithms are implemented using C++.

We use synthetic data streams in our experiments to gain easy control over the data distributions. We adopt parameters similar to those used in previous studies [7, 26]. The total number of different items in $\mathcal{I}$ is 1000, and the average size of transactions in $\mathcal{T}_t$ is 8. The number of transactions in each data stream is 100,000. Note that in real-world a data stream can be unbounded. However, none of the algorithms will be affected by the total number of transactions, as long as the stream is sufficiently large. Our tumbling windows are time-based, and the sizes of the windows are user determined based on the arrival rate of a data stream. We show in Section 5.5.2 how the window sizes affect our mining results. For ease of representation, we fix the transaction arrival rate for all data streams in this experiment, hence, the sizes of W_M and W_P can be represented using transaction counts.

5.2 Effectiveness over streams with stable distribution

In these experiments, we evaluate the effectiveness of the four algorithms over four data streams with Zipf-like distributions [27]. The lower the Zipf factor, the more evenly distributed are the data. A stream with higher Zipf factor is more skewed. Since FDPM and LC cannot deal with time-varying streams, to fairly compare effectiveness, the test data streams do not have distribution changes. The objective of these experiments is to test the performance of TWIM over streams with stable distribution.

The sizes of the two tumbling windows are $|W_M| = 500$ transactions, and $|W_P| = 1500$ transactions. The threshold values δ and θ are set to 0.8% and 0.5%, respectively. We discuss in Sections 5.3 and 5.5.1 how these thresholds affect the performance of these algorithms. The size of the sliding window used in SW is the same as the size of our W_M, i.e, 500 transactions. The error parameter and reliability parameter used in FDPM and LC are set to $\delta/10$ and 0.1, respectively[4]. According to earlier experiments, this setting will make FDPM and LC perform better [18, 26]. The recall (R) and precision (P) results are shown in Table 2.

Table 2. Recall and precision comparison

Stream	Zipf	TWIM		SW		FDPM		LC	
		R	P	R	P	R	P	R	P
D_1	0.8	0.68	1	0.71	0.74	0.69	1	1	0.52
D_2	1.2	0.87	1	0.79	0.83	0.80	1	1	0.62
D_3	2.0	0.93	1	0.92	0.95	0.95	1	1	0.84
D_4	2.8	1	1	1	1	1	1	1	0.88

These results demonstrate that, when the distribution of a data stream is near uniform, FDPM and SW perform slightly better than TWIM. However, when Zipf is higher, the performance of TWIM is comparable to FDPM and better than SW. When the stream is very skewed, TWIM, SW and FDPM can all find the exact answer. On the other hand, although LC always has a recall of 100%, its results are unreliable, especially for streams with lower Zipf. These results demonstrate that TWIM performs at least as well as existing algorithms on streams $without$ distribution change. Note that although the recall of FDPM is claimed to approach 1 at infinity [26], this only holds when the stream has no distribution change during its entire lifespan, which is a very strong and usually incorrect assumption for most real applications.

[4]The error parameter ϵ is used to control error bound. Smaller ϵ can reduce errors and increase the recall of FDPM and LC. The memory consumption of FDPM is reciprocal of the reliability parameter [26].

5.3 Effect of threshold δ

This set of experiments evaluate the effectiveness of the four algorithms with different values of threshold δ. For this set of experiments, we set $\theta = \delta - 0.3\%$. Note that threshold θ is mainly used to control the size of candidate list $\mathcal{C}$. As θ gets smaller, more candidate itemsets are selected, which leads to a higher memory consumption. On the other hand, when mining a time-varying data stream, a larger θ may cause TWIM's recall to decrease since there are fewer candidates. We demonstrate the effect of different θ values on mining time-varying streams in Section 5.5.1. In this set of experiments, since the testing data stream has a steady distribution, the size of $\mathcal{C}$ should not affect TWIM.

We apply TWIM, SW, FDPM and LC to data stream D_2 (as in Section 5.2) with Zipf 1.2, and vary δ from 0.4% to 2%. The results are shown in Table 3, which demonstrate that the effectiveness of TWIM is comparable with FDPM when δ varies. TWIM's recall is improved with higher δ. Although SW always has a better recall than TWIM and FDPM, its precision never reaches 1. LC has a low precision even when δ is high (2%).

Table 3. Results for varying δ value

δ	TWIM		SW		FDPM		LC	
	R	P	R	P	R	P	R	P
0.4%	0.62	1	0.76	0.57	0.65	1	1	0.44
0.8%	0.83	1	0.85	0.81	0.80	1	1	0.62
1.2%	0.94	1	1	0.87	0.93	1	1	0.74
2%	0.98	1	1	0.99	1	1	1	0.77

5.4 Effectiveness over dynamic streams

To evaluate the effectiveness of these three algorithms over time-varying data streams, we conducted several experiments.

We created two data streams D_5 and D_6 using the same statistics as in Section 4.1, with Zipf = 1.5 and 50,000 transactions in each stream. Both of the streams start changing their distributions every 10,000 transactions. The change of D_5 is steady and slow. It takes 4000 transactions for D_5 to complete its change. On the other hand, D_6 has a faster and more noticeable change: only 800 transactions to change. The sizes of the two tumbling windows are $|W_M| = 400$ transactions, and $|W_P| = 1500$ transactions. Threshold values δ and θ are 0.8% and 0.5%, respectively. The mining results after each distribution change for D_5 and D_6 are given in Tables 4 and 5.

Table 4. Mining results over D_5

change #	TWIM		SW		FDPM		LC	
	R	P	R	P	R	P	R	P
change 1	0.91	1	0.85	0.87	0.82	0.93	1	0.66
change 2	0.93	1	0.86	0.92	0.73	0.87	1	0.51
change 3	0.88	1	0.74	0.84	0.69	0.77	1	0.44
change 4	0.88	1	0.77	0.93	0.72	0.68	1	0.46
change 5	0.92	1	0.83	0.86	0.60	0.68	1	0.35

Table 5. Mining results over D_6

change #	TWIM		SW		FDPM		LC	
	R	P	R	P	R	P	R	P
change 1	0.95	1	0.72	0.82	0.87	0.82	1	0.58
change 2	0.97	1	0.71	0.77	0.78	0.81	1	0.51
change 3	0.93	1	0.69	0.80	0.65	0.74	1	0.38
change 4	1	1	0.74	0.71	0.67	0.66	1	0.41
change 5	0.88	1	0.71	0.89	0.53	0.64	1	0.32

These results show that TWIM and SW adapt to time-varying data streams, while neither FDPM nor LC is sensitive to distribution changes. The more severe the changes, the worse is the performance of FDPM and LC. Moreover, FDPM and LC's performance keeps worsening when more distribution changes occur in a stream, whereas TWIM and SW are not affected by the number of changes. SW performs worse than TWIM in both cases. Mining results of TWIM over the stream with faster and more noticeable distribution changes (D_6) are better than the one that changes slower (D_5), while SW seems more suitable to slower and mild changes. Note that as mentioned in Section 4.1, we may improve the mining results of TWIM for such slow-drifting data streams by reducing the sizes of W_M and W_P. We demonstrate the effect of different window sizes in Section 5.5.2.

5.5 TWIM Parameter Settings

5.5.1 *Effect of threshold θ*

We test TWIM on the time-varying streams D_5 and D_6 described in Section 4.4, and vary θ from 0.4% to 1%. The sizes of W_M and W_P are 400 transactions and 1500 transactions, respectively. Threshold value δ is fixed at 1.2%. The results are presented in Tables 6 and 7.

Table 6. Results for varying θ over D_5

Change #	θ (%)							
	0.4		0.6		0.8		1	
	R	P	R	P	R	P	R	P
change 1	0.96	1	0.97	1	0.88	1	0.72	1
change 2	0.95	1	0.95	1	0.83	1	0.74	1
change 3	0.93	1	0.89	1	0.85	1	0.68	1
change 4	0.98	1	0.94	1	0.88	1	0.75	1
change 5	0.89	1	0.87	1	0.74	1	0.69	1

Table 7. Results for varying θ over D_6

Change #	θ (%)							
	0.4		0.6		0.8		1	
	R	P	R	P	R	P	R	P
change 1	0.98	1	0.92	1	0.89	1	0.83	1
change 2	1	1	1	1	0.92	1	0.87	1
change 3	0.93	1	0.91	1	0.84	1	0.77	1
change 4	1	1	0.95	1	0.91	1	0.85	1
change 5	0.95	1	0.95	1	0.90	1	0.83	1

We see that the performance of TWIM can be improved by decreasing θ. However, as discussed in Section 4.2, a low θ value may result in higher memory consumption. The extreme case is $\theta = 0$. In this case, all infrequent itemsets will be treated as candidates, and thus the total number of counters is exponential.

5.5.2 *Varying window sizes*

To evaluate the effect of tumbling window sizes, we test TWIM on D_5 and D_6, and vary the size of W_M from 200 transactions to 1000 transactions, and W_P from 1000 transactions to 4000 transactions. Threshold values δ and θ are 0.8% and 0.5%, respectively. The experimental results are shown in Tables 8 and 9. Since the precision value is always 1, we only show the recall value.

We notice that larger windows size may reduce TWIM's recall, since sudden distribution changes will be missed. On the other hand, as mentioned in Section 4.1, large windows can ensure high accuracy of the estimated supports for candidate itemsets.

5.6 Memory usage

Table 8. Varying $|W_M|$ and $|W_P|$ over D_5

$\|W_M\|$	$\|W_P\|$	chg 1	chg 2	chg 3	chg 4	chg 5
200	1000	0.93	0.88	0.89	0.91	0.95
400	1500	0.87	0.92	0.85	0.88	0.90
600	2000	0.82	0.88	0.79	0.74	0.82
800	3000	0.75	0.73	0.72	0.67	0.69
1000	4000	0.68	0.72	0.66	0.64	0.61

Table 9. Varying $|W_M|$ and $|W_P|$ over D_6

$\|W_M\|$	$\|W_P\|$	chg 1	chg 2	chg 3	chg 4	chg 5
200	1000	0.99	0.97	0.93	0.95	0.88
400	1500	0.94	0.97	0.91	1	0.87
600	2000	0.89	0.92	0.85	0.89	0.86
800	3000	0.82	0.84	0.79	0.86	0.77
1000	4000	0.78	0.81	0.81	0.75	0.73

The major memory requirements for TWIM are the counters used for all items, frequent itemsets, and candidates. To reflect the memory usage of our approach, we report the maximal number of counters that we create for each experiment.

Table 10 presents the memory usage of TWIM, FDPM, and LC for mining data sets D_1, D_2, D_3 and D_4. Given that each counter takes 4 bytes, the memory requirement for mining these data streams using TWIM is around 60KB. According to this table, the memory consumed by SW is about four times of TWIM's memory usage. TWIM uses slightly more memory than FDPM, and LC has the lowest memory requirement.

Table 10. Maximal counters for mining D_1 - D_4

Stream	Maximal Counters			
	TWIM	SW	FDPM	LC
D_1	11892	47606	8478	7129
D_2	14533	59438	10128	8722
D_3	18002	71040	13502	11346
D_4	16115	56442	11764	10098

Table 11 shows TWIM's memory usage for the experiments in Section 5.5.1, demonstrating that its memory consumption is inversely correlated to threshold θ, and the maximum memory requirement is around 228KB for D_5 and 191KB for D_6.

To evaluate the effect of window sizes on memory usage, we present in Table 12 the maximum number of counters created for experiments in Section 5.5.2.

The maximum memory requirements for mining D_5 and D_6 are around 251KB and 225KB, respectively. We can see that larger windows sizes result in more counters to be used. Furthermore, the number of counters used for a stream with slow distribution changes is larger than the number of counters for a stream that changes fast.

5.7 CPU time analysis

Since TWIM is a window-based approach while neither FDPM nor LC use windows, it is hard to fairly compare their CPU times. However, to demonstrate that TWIM is efficient for high-speed data streams, we conducted a set of experiments.

By analyzing Algorithm 1, we can see that TWIM performs the largest amount of work when $|W_M|$ and $|W_P|$ tumble. Hence, we tested the average run time of TWIM at each tumble point for streams D_1 to D_6. The average run time for mining D_1 to D_6 are 3.3ms, 4.0ms, 2.5ms, 3.7ms, 5.3ms, and 5.9ms, respecitively. These results show that TWIM is an efficient algorithm suitable for online streams. We also notice that streams with distribu-

Table 11. Maximal counters when θ varies

Stream	θ (%)			
	0.4	0.6	0.8	1
Max Ctr.-D_5	64432	47210	36778	32002
Max Ctr.-D_6	51301	42676	35209	28123

Table 12. Maximum counters when $|W_M|$ and $|W_P|$ varies

$\|W_M\|$	$\|W_P\|$	max Ctr. - D_5	max Ctr. - D_6
200	1000	42398	39901
400	1500	50006	44872
600	2000	56020	51922
800	3000	59891	54646
1000	4000	65335	59043

tion changes (D_5 and D_6) require slightly longer processing time, since $\mathcal{A}_{\mathcal{T}_t}$ and $\mathcal{C}$ are updated more frequently.

6. Conclusion

In this paper, we propose a novel algorithm called TWIM for mining frequent itemsets. Our approach has the ability to detect changes in a data stream and update mining results in real-time. We use two tumbling windows to maintain current frequent itemsets and predict distribution changes. A list of candidate itemsets is generated and updated during mining. The candidates are the itemsets that have the potential to become frequent if distribution changes. Every time the two tumbling windows move, we apply a set of heuristics to update the candidate list and maintain frequent itemsets. Candidates that become frequent are moved to the frequent itemset list, new candidates are added, and itemsets that no longer have supports greater than threshold value θ are removed. Unlike most existing algorithms that are false-positive oriented, our approach produces only true frequent itemsets, and requires less memory. Experimental results demonstrate that TWIM has promising performance on mining data streams with or without distribution changes.

We are currently investigating a number of issues, including proving the complexity for finding the kth frequent itemset in a data stream, developing more heuristics for maintaining candidate itemsets, designing a more sophisticated and more efficient counting system, and analyzing the relationship among thresholds, window sizes, and memory space for different applications.

7. References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 487–499, 1994.

[2] B. Babock, S. Babu, M. Datar, R. Motiwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 1–16, 2002.

[3] J. Chang and W. Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 487–492, 2003.

[4] J. Chang and W. Lee. A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineering*, pages 753–762, 2004.

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. Int. Colloquium on Automata, Languages, and Programming*, pages 693–703, 2002.

[6] J. Cheng, Y. Ke, and W. Ng. Maintaining frequent itemsets over high-speed data streams. In *Proc. Pacific-Asia Conf. on Knowledge Discovery and Data Mining PAKDD*, pages 462–467, 2006.

[7] Y. Chi, H. Wang, P. Yu, and R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proc. 2004 IEEE Int. Conf. on Data Mining*, pages 59–66, 2004.

[8] Cormode and Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 296–306, 2003.

[9] Demaine, Lopez-Ortiz, and Munro. Frequency estimation of internet packet streams with limited space. In *Proc. 10th Annual European Symposium on Algorithms*, pages 348–360, 2002.

[10] M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, (2):18–26, 2005.

[11] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, and R. Sharma. Discovering all most specific sentences. *ACM Trans. Database Sys.*, (2):140–174, 2003.

[12] M. Halatchev and L. Gruenwald. Estimating missing values in related sensor data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 83–94, 2005.

[13] G. Hulten, L. Spencer, and P. Domingos. Mining time-chaning data streams. In *Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[14] N. Jiang and L. Gruenwald. Research issues in data stream association rule mining. *ACM SIGMOD Record*, (1):14–19, 2006.

[15] R. Jin and G. Aggrawal. Efficient decision tree constructions on streaming data. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 571–576, 2003.

[16] R. Karp, C. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in sets and bags. *ACM Trans. Database Sys.*, pages 51–55, 2003.

[17] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 180–191, 2004.

[18] Manku and Motwani. Approximate frequency counts over data streams. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 346–357, 2002.

[19] D. Cai et. al. Maids: Mining alarming incidents from data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 919–920, 2004.

[20] D. Carney et. al. Monitoring streams - a new class of data management application. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 215–226, 2002.

[21] F. Angiulli et. al. On the complexity of inducing categorical and quantitative association rules. *Theoretical Computer Science*, pages 217–249, 2004.

[22] J. Wang et. al. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 236–245, 2003.

[23] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, (3):410–421, 1979.

[24] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 226–235, 2003.

[25] L. Yang and M. Sanver. Mining short association rules with one database scan. In *Proc. Int. Conf. on Information and Knowledge Engineering*, 2004.

[26] 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. 30th Int. Conf. on Very Large Data Bases*, pages 204–215, 2004.

[27] G. K. Zipf. *Human behavior and the principle of least-effort*. Addison-Wesley, 1949.