Mining Frequent Itemsets in Time-Varying Data Streams

Yingying Tao and M. Tamer Özsu
University of Waterloo
Waterloo, Ontario, Canada
{y3tao, tozsu}@cs.uwaterloo.ca

ABSTRACT

Mining frequent itemsets in data streams 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. 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.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications – Data Mining

General Terms

Algorithms, Performance, Experimentation

Keywords

Data stream, Frequent itemset

1. INTRODUCTION

Mining frequent itemsets in data stream applications is beneficial for a number of purposes such as knowledge discovery, trend learning, fraud detection, transaction prediction and estimation. 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 [4].

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, data streams can change over time. Hence, an itemset that was once infrequent can become frequent if a stream changes its distribution. It is hard to detect such itemsets that change from infrequent to frequent, since they are not maintained due to limited memory. 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.

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.

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.

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.

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* $\mathcal{A}$.

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 $\mathcal{A}$.

3. RELATED WORKS

Mining frequent items and itemsets is a challenging task and has attracted attention in recent years. Jiang and Gruenwald [5] provide a good review of research issues in frequent itemsets and association rule mining over data streams. 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. However, 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. Most of the techniques proposed in literature are false-positive oriented. False-positive techniques may consume more memory, and are not suitable for many applications where accurate results, even if not complete, are preferred.

4. TWIM ALGORITHM

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.

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.

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. 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.

Figure 1 demonstrates the relationship between W_M and W_P. In Figure 1, W_M and W_P are the windows before tumbling, while W'_M and W'_P are windows afterwards. 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.

Mining frequent itemsets requires keeping counters for all itemsets; however, the number of itemsets is exponential.

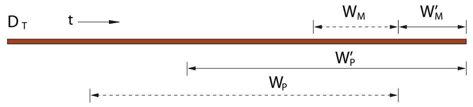

Figure 1: Tumbling windows for a data stream

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.
- 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*. 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 is reasonable.

4.1 Predicting candidates

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 memory since more candidates are generated; a high θ value can reduce memory usage by sacrificing the number of resulting frequent itemsets.

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. Many existing frequent itemset mining techniques for streams are derived from the popular Apriori algorithm [1]. Apriori increases the size of candidate supersets by 1 at every run, until the largest itemset is detected. This strategy successfully reduces the number of candidates; however, in cases when the itemset size $|\mathcal{I}|$ is large, it may take extremely long time until one large frequent itemset is detected. Furthermore, since Apriori-like approaches only check the supersets of the existing frequent itemsets, the subsets of existing frequent itemsets are not considered. To solve these problems, we introduce the concept of *smallest cover set* defined as follows.

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 $\mathcal{A}^C$.

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 $i = 1, ..., q$, we call $\mathcal{A}^C_s$ the *smallest cover set* of $\mathcal{A}$, denoted as $\mathcal{A}^{SC}$.

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.

4.1.1 *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 $\mathcal{A}_{\mathcal{T}_t}$. Also detect any new frequent items and add them into $\mathcal{A}_{\mathcal{T}_t}$.
- **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.
- **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 $\mathcal{C}$. This process can be regarded as the reverse process of 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.

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 $|\mathcal{C}_{\mathcal{A}}| + \frac{2}{|W_M|}\beta \le 2p - k$, 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 $\mathcal{A}^{SC}_{\mathcal{T}_t}$. 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.

4.1.2 *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 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 $\mathcal{B}_k$.
- **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.

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.

4.1.3 *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 $\mathcal{C}$ for a long time, if it becomes frequent at time t_i, its support may not be greater than 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 $\mathcal{A} \in \mathcal{C}$. Since W_M and W_P are time-based tumbling windows, they tumble every $|W_M|$ time units. 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 $\mathcal{A}$. When W_M and W_P tumbles, $sup(\mathcal{A})$ is updated by expiring transactions before chk_1.

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 $\mathcal{C}$, 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.

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 $\mathit{offset}(\mathcal{A})$, 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 $S(\mathcal{A}) = \mathit{sup}(\mathcal{A})/(N_{t'} - \mathit{offset}(\mathcal{A}))$, where $N_{t'}$ is the total number of transactions received at time t', as the data stream monotonically grows.

5. EXPERIMENTS

We conduct a series of experiments to evaluate TWIM's performance in comparison with three others: SW method [2], which is a sliding window based technique suitable for dynamic data streams, FDPM [7], which is also a false-negative algorithm, and Lossy Counting (LC) [6], which is a widely-adopted false positive algorithm. 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 [3, 7]. 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.

5.1 Effectiveness over non-dynamic streams

We evaluate the effectiveness of the four algorithms over four data streams with Zipf-like distributions. 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 results indicate that, that TWIM performs at least as well as existing algorithms on streams *without* distribution change.

5.2 Effect of threshold δ

To evaluate the effectiveness of the four algorithms with different values of threshold δ, we apply TWIM, SW, FDPM and LC to a data stream with Zipf 1.2, and vary δ from 0.4% to 2%. The results 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%).

5.3 Effectiveness over dynamic streams

To evaluate the effectiveness of these three algorithms over time-varying data streams, we created two data streams D_s and D_f 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_s is steady and slow, whereas D_f has a faster and more noticeable change.

The results show that TWIM and SW adapt to time-varying data streams, while neither FDPM nor LC is sensitive to distribution changes. SW performs worse than TWIM in both tests. Mining results of TWIM over the stream with faster and more noticeable changes are better than the one that changes slower, while SW seems more suitable to slower and mild changes. However, note that we may improve the mining results of TWIM for such slow-drifting data streams by reducing the sizes of W_M and W_P.

5.4 TWIM Parameter Settings

We test TWIM on D_s and D_f and vary θ from 0.4% to 1%. It is shown that the performance of TWIM can be improved by decreasing θ. However, a low θ value may result in higher memory consumption.

To evaluate the effect of tumbling window sizes, we vary the size of W_M from 200 transactions to 1000 transactions, and W_P from 1000 transactions to 4000 transactions, and test TWIM on these two streams. We notice that larger windows size may reduce TWIM's recall, since sudden distribution changes will be missed. On the other hand, large windows can ensure high accuracy of the estimated supports for candidate itemsets.

5.5 Memory usage

The major memory requirements for TWIM are the counters used for all items, frequent itemsets, and candidates. We compare the maximal number of counters that we create for each of the previous experiments. According to the results, 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. We also notice that the memory consumption is inversely correlated to threshold θ, and larger windows sizes result in more counters to be used.

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 that have the potential to become frequent if distribution changes is generated and updated during mining. Every time the two tumbling windows move, we apply a set of heuristics to update the candidate list and maintain frequent itemsets. 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] 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.

[3] 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.

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

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

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

[7] 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.