A Sliding-window Based Adaptive Approximating Method to Discover Recent Frequent Itemsets from Data Streams

Kuen-Fang Jea and Chao-Wei Li

***Abstract* — Frequent-patterns discovery in data streams is an active research area and more challenging than traditional database mining since several additional requirements need to be satisfied. In this paper, we propose a mining algorithm for finding frequent itemsets over sliding windows in a data stream. Different from most existing algorithms, our method is based on the theory of Combinatorial Approximation to approximate the counts of itemsets from some summary information of the stream. We also devise the novel concept of fair-cutter, which results in an original technique called dynamically approximating and makes our method capable of approximating adaptively for different itemsets. Empirical results show that the proposed method is quite efficient and scalable; moreover, the mining result from approximations achieves high accuracy.**

***Index Terms* — Combinatorial approximation, data stream, data-stream mining, frequent itemset, and sliding window.**

I. Introduction

In many application domains, data is presented in the form of *data streams* which originate at some endpoint and are transmitted through the communication channel to the central server. Some well-known examples include market basket, traffic signals, web-click packets, ATM transactions, and sensor networks. In these applications, it is desirable that we obtain some useful information, like patterns occurred frequently, from the streaming data, to help us make some advanced decisions. *Data-stream mining* is such a technique that can find valuable information or knowledge from a great deal of primitive data.

Data-stream mining differs from traditional data mining since its input of mining is data streams, while the latter focuses on mining (static) databases. Compared to traditional databases, mining in data streams has more constraints and requirements [1]. First, each element (e.g., transaction) in the data stream can be examined only once or twice, making traditional multiple-scan approaches infeasible. Second, the consumption of memory space should be confined in a range, despite that data elements are continuously streaming into the local site. Third, notwithstanding the data characteristic of incoming stream may be unpredictable, the mining task should proceed normally and offer acceptable quality of results. Fourth, the latest analysis result of the data stream should be available as soon as possible when the user invokes a query. As a result, one good stream mining algorithm needs to possess efficient performance and high throughput, while slight approximate errors occurred in the mining result is usually acceptable by the user.

The research of data-stream mining in general can be classified into three categories according to the stream processing model [2]. The first one is the *landmark window model*. In this model, there is a time point called the landmark, and the range of mining contains all the data elements (or transactions) between the landmark and current point. The second one is the *damped* (*fading*/*decay*) *window model*. In this model, each element is associated with a weight which is relative to the time. When a stream element is just arriving, its weight is at the highest value and then will get damped continuously as time goes by. The last one is the *sliding window model*. In this model, the range of mining is confined to the elements contained in a window which will slide with time. The window always covers a certain number of most recent elements and the mining task focuses on these elements at any point.

In this paper, we propose a remarkable approximating method for discovering *frequent itemsets* in a transactional data stream under the sliding window model. Based on a theory of *Combinatorial Mathematics*, the proposed method approximates the counts of itemsets from certain recorded summary information without scanning the input stream for each itemset. Together with an innovative technique called *dynamically approximating* to select parameter-values properly for different itemsets to be approximated, our method is adaptive to streams with different distributions. Through the experimental results, we found that our method has efficient performance with pretty accurate mining result.

The rest of this paper is organized as follows. Section II briefly introduces the related research on data-stream mining in recent years. In Section III we state our problem and give the necessary representation of symbols. Section IV describes the way our method processes on a data stream with a sliding window, and explains the technique to approximate dynamically. Next in Section V, we give an explanation of the proposed algorithm. Section VI reports the experimental results of our method with simple analyses. Finally, Section VII concludes this paper.

II. Related Work

There are a number of research works which study the problem of data-stream mining in the first decade of 21st

The authors are with the Department of Computer Science and Engineering, National Chung-Hsing University, Taichung 402, Taiwan, R.O.C. (e-mail: kfjea@cs.nchu.edu.tw; s9656026@cs.nchu.edu.tw).

century. Among these studies, *Lossy Counting* [3] is the most famous method of mining *frequent itemsets* (FIs) through data streams under the landmark window model. Besides the user-specified *minimum support threshold* (*ms*), Lossy Counting also utilizes an *error parameter*, ε, to maintain those infrequent itemsets having the potential to become frequent in the future. With the use of ε, when an itemset is newly found, Lossy Counting knows the upper bound of counts that itemsets may have (in the previous stream data) before it has been monitored by the algorithm.

Based on the ε mechanism of Lossy Counting, [4] proposed the *sliding window* method, which can find out frequent itemsets in a data stream under the sliding window model with high accuracy. The sliding window method processes the incoming stream data transaction by transaction. Each time when a new transaction is inserted into the window, the itemsets contained in that transaction are updated into the data structure incrementally. Next, the oldest transaction in the original window is dropped out, and the effect of those itemsets contained in it is also deleted. The sliding window method also has a periodical operation to prune away unpromising itemsets from its data structure, and the frequent itemsets are output as mining result whenever a user requests.

In the sliding window model, there are two typical mining methods: *Moment* [5] and *CFI-Stream* [6]. Both the two methods aimed at mining *Closed Frequent Itemsets* (CFIs), a complete and non-redundant representation of the set of FIs. Moment uses a data structure called *CET* to maintain a dynamically selected set of itemsets, which includes CFIs and itemsets that form a boundary between CFIs and the rest of itemsets. The CFI-Stream algorithm, on the other hand, uses a data structure called *DIU tree* to maintain nothing other than all *Closed Itemsets* over the sliding window. The current CFIs can be output anytime based on any *ms* specified by the user.

Besides, there are still some interesting research works [9] [10] [11] on the sliding window model. In [9] a false-negative approach named *MineSW* was proposed. By employing a progressively increasing function of ms, MineSW greatly reduces the number of potential itemsets and would approximate the set of FIs over a sliding window. In [10] a data structure called *DSTree* was proposed to capture information from the streams. This tree captures the contents of transactions in a window, and arranges tree nodes according to some canonical order. The DSTree can be easily maintained and used to discover frequent itemsets. In [11] a *verification* algorithm was introduced to improve the performance of mining for *association rules*, and thus an FI mining method was proposed whose running time is nearly constant with respect to the window size.

In recent two years, a new kind of data-stream mining method named *DSCA* has been proposed [12]. DSCA is an approximate approach based on the application of the *Principle of Inclusion and Exclusion* in *Combinatorial Mathematics* [7]. One of the most notable features of DSCA is that it would approximate the count of an arbitrary itemset, through an equation (i.e., Equation (4) in [12]), by only the sum of counts of the first few orders of its subsets over the data stream. There are also two techniques named *counts bounding* and *correction*, respectively, integrated within DSCA. Both techniques are of the purpose to improve the quality of DSCA's approximation, while they adopt different means to achieve the purpose. By working together with these original techniques, the mining result of DSCA reaches good accuracy.

The concept of *Inclusion and Exclusion Principle* [7] is valuable that it may also be applied in mining FIs under different window models other than the landmark window. Based on the theory of *Approximate Inclusion–Exclusion* [8], we devise and propose a novel algorithm, called *SWCA*, to approximate dynamically and discover FIs over the sliding window in a transactional data stream.

III. Problem Description

Let $I = \{x_1, x_2, \ldots, x_z\}$ be a set of items (or attributes). An itemset (or a pattern) X is a subset of I and written as $X = x_i x_j \ldots x_m$. The length (i.e., number of items) of an itemset X is denoted by $|X|$. A transaction, T, is an itemset and T supports an itemset, X, if $X \subseteq T$. A transactional data stream is a sequence of continuously incoming transactions. A segment, S, is a sequence of fixed number of transactions, and the size of S is indicated by s. A window, W, in the stream is a set of successive w transactions, where $w \geq s$. A sliding window in the stream is a window of a fixed number of most recent w transactions which slides forward for every transaction or every segment of transactions. We adopt the notation I_l to denote all the itemsets of length l together with their respective counts in a set of transactions (e.g., over W or S). In addition, we use T_n and S_n to denote the latest transaction and segment in the current window, respectively. Thus, the *current window* is either $W = < T_{n-w+1}, \ldots, T_n >$ or $W = < S_{n-m+1}, \ldots, S_n >$, where w and m denote the size of W and the number of segments in W, respectively.

In this research, we employ a *prefix tree* which is organized under the *lexicographic order* as our data structure, and also processes the growth of itemsets in a lexicographic-ordered way. As a result, an itemset is treated a little bit like a *sequence* (while it is indeed an itemset). A superset of an itemset X is the one whose length is above $|X|$ and has X as its *prefix*. We define $\text{Growth}_l(X)$ as the set of supersets of an itemset X whose length are l more than that of X, where $l \geq 0$. The number of itemsets in $\text{Growth}_l(X)$ is denoted by $|\text{Growth}_l(X)|$.

We adopt the symbol $\text{cnt}(X)$ to represent the count-value (or just count) of an itemset X. The count of X over W, denoted as $\text{cnt}_W(X)$, is the number of transactions in W that support X. So $\text{cnt}_S(X)$ represents the count of X over a segment S. Given a user-specified *minimum support threshold* (*ms*), where $0 < ms \leq 1$, we say that X is a *frequent itemset* (FI) over W if $\text{cnt}_W(X) \geq ms \times w$, otherwise X is an *infrequent itemset* (IFI). The FI and IFI over S are defined similarly to those for W.

Given a data stream in which every incoming transaction has its items arranged in order, and a *changeable value* of *ms* specified by the user, the problem of mining FIs over a sliding window in the stream is to find out the set of frequent itemsets over the window at different slides.

We remark that most of the existing stream mining methods [3] [4] [5] [9] work with a basic hypothesis that they know the user-specified *ms* in advance, and this parameter will remain unchanged all the time before the stream terminates. This hypothesis may be unreasonable, since in general, a user may wish to tune the value of *ms* each time he/she makes a query for the purpose of obtaining a more preferable mining result. An unchangeable *ms* leads to a serious limitation and may be impractical in most real-life applications. As a result, we relax this constraint in our problem that the user is allowed to change the value of *ms* at different times, while our method must still work normally.

The following *approximate equation*, which is derived from the equation of *Approximate Inclusion–Exclusion* in [8], is the core (or the basis) of our approximating approach.

$$|A_1 \cup A_2 \cup \ldots \cup A_m| \doteqdot \sum_{|S|\le k} \alpha_{|S|}^{k,m} |\bigcap_{i\in S} A_i|. \qquad (1)$$

According to (1), given a collection of *m* sets, $A_1, A_2, \ldots, A_m$, the size of *m*-union term can be approximated even if we know only the sizes of partial intersection terms for some *k*, where $k<m$. By considering each item as a *set*, and the number of its occurrences in transactions (i.e. count) as its corresponding *size*, we could indirectly apply (1) to *approximate* the count of an itemset from the sum of counts of its subsets of different lengths. For example, if we have the information called the *summary* in Table 1, we would approximate the count of a 3-itemset, such as *abc*, through (1), based on the sum of counts of its 1-subsets (i.e., *a*, *b*, and *c*) and 2-subsets (i.e., *ab*, *ac*, and *bc*), both of which can be obtained from Table 1.

Table 1. An example of some itemsets with counts

Itemset	Count	Itemset	Count
a	25	*ab*	7
b	17	*ac*	6
c	20	*ad*	6
d	13	*bc*	8
		bd	7
		cd	5

If we have the summary of I_1 and I_2 of a stream over the sliding window, we could achieve the mining work anytime when requested by approximating the counts of itemsets, through applying (1) with the parameter setting k=2, based on the summary information we have kept. Now we outline the way how we process the incoming transactions of the data stream. For each incoming transaction *T*, we find all the first-two-order lengths of subsets contained in *T* and record their occurrences in our data structure. As a result, the first-two-order summary (i.e., I_1 and I_2) of the stream is maintained. We say that the *summary length* in our method is 2. The manner of incrementally updating the summary over the sliding window in a stream will be detailed in Section 4.

The aforementioned means is the primary concept to approximate the counts of itemsets. In practice, for an itemset *X* to be approximated, the count of each of *X*'s subset may come from (be supported by) various transactions where many of them may have insufficient relation to *X*. Using the original count-values of subsets to approximate the count of *X* may sometimes bring about considerable error. For a better approximation, there is a possible way, which is to *bound* the range of counts of subsets for the itemset to be approximated.

Let *Y* be a 3-itemset (to be approximated) and *y* be a 1-subset of *Y*. To obtain a better approximation of *Y*, the count-value to each subset *y* with respect to *Y* has a particular range, which can be determined by *Y*'s 2-subsets that have *y* as their common subset, respectively. This range of *y*'s count is bounded by an *upper bound* (i.e., the maximum) and a *lower bound* (i.e., the minimum), and count-values within this range are the set of portions of *y*'s original count which is more relevant for *y* with respect to *Y*. We define $\mathrm{Scb}_y(Y)$ as the set of count-values of *y* with respect to *Y* within the range obtained through the aforesaid manner. Besides, the upper bound and the lower bound of count-values of *y* are denoted by $\mathrm{ub}_y(Y)$ and $\mathrm{lb}_y(Y)$, respectively.

We use the same example in Table 1 to illustrate this concept. To approximate an itemset *abc*, the count of its 1-subset *a* with respect to *abc* can be bounded by *abc*'s 2-subsets *ab* and *ac*, which have *a* as their common subset. In Table 1, the counts of *ab* and *ac* are 7 and 6, respectively, and therefore the count of *a* with respect to *abc* is bounded within the range between the lower bound of 7 (i.e., the one of *ab* and *ac* with greater count) and the upper bound of 13 (i.e., the sum of counts of *ab* and *ac*). We have $\mathrm{Scb}_a(abc) = \{7, \ldots, 13\}$, $\mathrm{ub}_a(abc) = 13$, and $\mathrm{lb}_a(abc) = 7$. Compared with *a*'s original count of 25, count-values in the set $\mathrm{Scb}_a(abc)$ are the portions of 25 which are relevant for *a* with respect to *abc*. The ranges of counts of other 1-subsets *b* and *c* of *abc*, i.e., $\mathrm{Scb}_b(abc)$ and $\mathrm{Scb}_c(abc)$, can be obtained similarly as we do for *a*.

By using a count-value in the range of each 1-subset to form the sum of 1-subset term, we could then obtain a better approximation for an itemset than that of using the original counts of 1-subsets. However, for each 1-subset, we have no idea about what count-value in its range is better than others. For example, in the above instance of *a*, even if we obtain a range of counts in $\mathrm{Scb}_a(abc)$, we have no idea about which value in $\mathrm{Scb}_a(abc)$, e.g., $\mathrm{ub}_a(abc)$ or $\mathrm{lb}_a(abc)$, is more appropriate than others for *a* to approximate *abc*. As a result, our main issue (and also contribution) in this research is to devise a mechanism for the above problem. That is, the mining algorithm needs to be able to choose the appropriate count-values for the 1-subsets when approximating a 3-itemset *Y*, and thus can approximate *dynamically* for different 3-itemsets. Before we illustrate our means in the next section, the following property is first introduced.

Lemma 1 *Let* $\mathrm{cnt}^{\mathrm{ub}}(Y)$ *and* $\mathrm{cnt}^{\mathrm{lb}}(Y)$ *respectively be the approximate counts of Y obtained by using* $\mathrm{ub}_y(Y)$ *and* $\mathrm{lb}_y(Y)$ *of count-values to every 1-subset* $y \in Y$ *during the approximating process. Then* $\mathrm{cnt}^{\mathrm{ub}}(Y) \le \mathrm{cnt}^{\mathrm{lb}}(Y)$.

Proof Let T^{ub} and T^{lb} be the sums of counts of 1-subsets of *Y* obtained by choosing $\mathrm{ub}_y(Y)$ and $\mathrm{lb}_y(Y)$ for each $y \in Y$, respectively. Then $\mathrm{T}^{\mathrm{ub}} \ge \mathrm{T}^{\mathrm{lb}}$ since $\mathrm{ub}_y(Y) \ge \mathrm{lb}_y(Y)$ for each *y*. According to (1) with the parameters setting m=3 and k=2, we can eventually obtain the following simplified equation: $\mathrm{cnt}(Y) \doteqdot (1-\alpha_2^{2,3})\, c + (\alpha_1^{2,3}-1)\, d$, where the symbol $\alpha_j^{k,m}$ denotes the coefficient of linearly transformed *Chebyshev polynomial*, and *c* and *d* represent the sum of counts of *Y*'s 2-subsets and that of counts of *Y*'s 1-subsets, respectively. Since the value of $\alpha_1^{2,3}$ is less than 1, the coefficient $(\alpha_1^{2,3}-1)$ for 1-subset term is then negative, which means that the value of 1-subset term will be subtracted from the other term. Thus, by substituting T^{ub} and T^{lb} for *d* respectively in the approximate equation and knowing that $\mathrm{T}^{\mathrm{ub}} \ge \mathrm{T}^{\mathrm{lb}}$, we have $\mathrm{cnt}^{\mathrm{ub}}(Y) \le \mathrm{cnt}^{\mathrm{lb}}(Y)$.

IV. Sliding-window Processing

In research works under the sliding window model [4] [5] [6], the sliding of window is handled transaction by transaction; however, we have a different opinion. Unlike the landmark window model, transactions in the sliding window model will be both inserted into and dropped out from the window. The transaction-by-transaction sliding of a window leads to excessively high frequency of processing. In addition, since the transit of a data stream is usually at a high speed, and the impact of one single transaction to the entire set of transactions (in the current window) is very negligible, making it reasonable to handle the window sliding in a wider magnitude. Therefore, for an incoming transactional data stream to be mined, we propose to process on a *segment-oriented window sliding*.

We conceptually divide the sliding window further into several, say, m, segments, where the term *segment* is the one we have defined earlier in Section 3. Each of the m segments contains a set of successive transactions and is of the same size s (i.e., contains the equal number of s transactions). Besides, in each segment, the summary (which contains I_1, I_2, and the *fair-cutters* which we will introduced later) of transactions belonging to that segment is stored in the data structure we use. We call the sliding of window "segment in-out," which is defined as follows.

Definition 1 (**Segment in-out**) Let S_c denote the current segment which is going to be inserted into the window next (after it is full of s transactions). A *segment in-out* operation (of the window) is that we first insert S_c into and then extract S_{n-m+1} from the original window, where n denotes the id of latest segment in the original window. Therefore, the windows before and after a sliding are $W = < S_{n-m+1}, \ldots, S_n >$ and $W = < S_{n-m+2}, \ldots, S_n, S_c >$, respectively.

By taking this segment-based manner of sliding, each time when a segment in-out operation occurs, we delete (or drop out) the earliest segment, which contains the summary of transactions of that segment, from the current window at each sliding. As a result, we need not to maintain the whole transactions within the current window in memory all along to support window sliding. In addition, we remark that the parameter m directly affects the consumption of memory. A larger value of m means the window will slide (update) more frequently, while the increasing overhead of memory space is also considerable. In our opinion, an adequate size of m that falls in the range between 5 and 20 may be suitable for general data streams.

Theorem 1 *For a 2-itemset X and a threshold ms, let* $TP^{ub}(X)$ *and* $TP^{lb}(X)$ *be the true-positive rates of X's 3-supersets in the mining result resulting from adopting Ubc and Lbc to all the 1-subsets of each superset, respectively. Then we have* $TP^{ub}(X) \leq TP^{lb}(X)$.

Proof Let P be the number of FIs of X's 3-supersets with respect to *ms*. Also, let P^{ub} and P^{lb} respectively be the numbers of true FIs of X's 3-supersets found by choosing Ubc and Lbc to 1-subsets. According to Lemma 1, we have $cnt^{ub}(Y) \leq cnt^{lb}(Y)$ for each 3-superset Y of X, which means that the number of true-positive itemsets (i.e., FIs) obtained by choosing Lbc is at least equal to that of choosing Ubc, i.e., $P^{ub} \leq P^{lb}$. Since $TP^{ub}(X) = P^{ub}/P$ and $TP^{lb}(X) = P^{lb}/P$, we then have $TP^{ub}(X) \leq TP^{lb}(X)$.

Theorem 2 *For a 2-itemset X and a threshold ms, let* $TN^{ub}(X)$ *and* $TN^{lb}(X)$ *be the true-negative rates of X's 3-supersets in the mining result resulting from adopting Ubc and Lbc to all the 1-subsets of each superset, respectively. Then we have* $TN^{ub}(X) \geq TN^{lb}(X)$.

Proof Let N be the number of IFIs of X's 3-supersets with respect to *ms*. Also, let N^{ub} and N^{lb} respectively be the numbers of true IFIs of X's 3-supersets determined by choosing Ubc and Lbc to 1-subsets. According to Lemma 1, we have $cnt^{ub}(Y) \leq cnt^{lb}(Y)$ for each 3-superset Y of X, which means that the number of true-negative itemsets (i.e., IFIs) determined by choosing Ubc is at least equal to that of choosing Lbc, i.e., $N^{ub} \geq N^{lb}$. Since $TP^{ub}(X) = N^{ub}/N$ and $TP^{lb}(X) = N^{lb}/N$, we then have $TP^{ub}(X) \geq TP^{lb}(X)$.

Now we discuss the issue of selecting suitable count-values in the bounded range of subsets for approximating an itemset. According to Lemma 1 in Section 3, using the *lower bound of counts* (Lbc) for 1-subsets always results in a higher approximate count for an itemset than that of using the *upper bound of counts* (Ubc). It follows from Theorem 1 and Theorem 2 that, adopting Lbc to the 1-subsets to approximate the 3-supersets of an 2-itemset X will reach a higher *true-positive rate* (i.e., *recall ratio*) in the result than that of using Ubc, while choosing Ubc to the 1-subsets will obtain a higher *true-negative rate* (which usually concerns a higher *precision ratio*) in the result than that of using Lbc. However, it is actually unknown whether to adopt Ubc or Lbc to 1-subsets for approximating an itemset Y.

To tackle the problem that the proper count-values, say, Lbc or Ubc, of the ranges (bounded through the means we described in Section 3) of an itemset's 1-subsets are unable to be determined dynamically, we introduce the concept of "fair-cutter," which will support our method to dynamically approximate and also make Theorem 1 and Theorem 2 applicable. Briefly, the fair-cutter for a 2-itemset X is a count-value which can tell us whether the *majority* of X's 3-supersets' counts are above *ms* or not. We formally define the fair-cutter of an itemset as follows.

Definition 2 (**Fair-cutter**) Let X be an itemset and P be the set of itemsets belonging to $\text{Growth}_1(X)$ whose counts are greater than 0. That is, $P = \{Y \mid Y \in \text{Growth}_1(X) \wedge \text{cnt}(Y) > 0\}$. The number of members in P is denoted by $|P|$. The *fair-cutter* (FC) of X for its supersets in P, denoted as FC(X), is the *minimum count-value ct* such that (1) $ct \leq \text{cnt}(X)$, and (2) the number of itemsets whose counts are greater than or equal to ct and that of itemsets whose counts are less than or equal to ct are above $1/2 \times |P|$.

According to Definition 2, in all the itemsets belonging to $\text{Growth}_1(X)$ whose counts are not 0 (i.e., P), both the numbers of supersets whose counts are greater than or equal to FC(X), and those whose counts are less than or equal to FC(X), are above half of that in the entire set of P. That is, for the above-mentioned two sets determined by FC(X), they have the common itemset(s) whose count(s) equal(s) FC(X), and their sizes reach the majority of P, respectively. As the name of fair-cutter indicates, the sizes of the two sets of P generated by FC(X) are *balanced* or *near-balanced*.

Example 1 Table 2 records a 2-itemset *ak* and all the 3-itemsets which can be grown from *ak*. The number in the parentheses behind each itemset represents its count. In this example, Growth$_1$(*ak*) = {*akl*, …, *aky*, *akz*}, |Growth$_1$(*ak*)| = 15, P = {*akl*, *akm*, …, *akv*, *akw*} (that is, the itemsets in Growth$_1$(*ak*) whose counts are greater than 0), and |P| = 12. From Definition 2, the FC of *ak* (i.e., FC(*ak*)) is 5 since the number of itemsets in P whose counts are above 5 is six, and the number of itemsets in P whose counts are below 5 is seven, both of these two values are above 1/2×|P|. Besides, the value "5" is the minimum count that satisfies the above condition.

Table 2. A 2-itemset and its 3-supersets

ak (10)				
akl (3)	*akm* (6)	*akn* (2)	*ako* (6)	*akp* (3)
akq (7)	*akr* (4)	*aks* (1)	*akt* (7)	*aku* (9)
akv (8)	*akw* (5)	*akx* (0)	*aky* (0)	*akz* (0)

Having the aforementioned concept of FC, we now explain how to select the appropriate counts from Ubc and Lbc to 1-subsets for approximating the count of an itemset Y. According to Lemma 1, we know that choosing Lbc to 1-subsets will result in higher approximations, make more itemsets exceeding *ms*, thus produce more frequent itemsets in the result than that of choosing Ubc. In contrast, adopting Ubc to 1-subsets will result in lower approximations for itemsets, which leads to less frequent itemsets in the result than that of adopting Lbc.

Given a user-specified *ms* and a 2-itemset X, if the fair-cutter of X (i.e., FC(X)) is above *ms*, according to Definition 2, we know that the majority of members in P are over the minimum support threshold, i.e., they are frequent itemsets. Oppositely, if FC(X) is below *ms*, we understand that the majority of members in P are under *ms*, i.e., they are infrequent itemsets. It is known that a frequent itemset (in the mining result) will raise the *true-positive rate*, while an infrequent itemset will diminish the *true-negative rate* and also the *precision*. By taking both Theorem 1 and Theorem 2 into account, this gives us a heuristic to choose Lbc or Ubc dynamically. In the former case, which the frequent itemsets are the majority, we should adopt Lbc to 1-subsets for approximating X's 3-supersets (to produce more true-positive itemsets). In the latter case, where the infrequent itemsets become the majority, we should instead adopt Ubc to 1-subsets for approximating X's 3-supersets (to produce less false-positive itemsets). By comparing *ms* with the respective FCs of itemsets, we would determine the better ways to choose the counts of 1-subsets for different itemsets to be approximated. We formally name this technique *dynamically approximating*.

V. The SWCA Algorithm

Based on the previous analysis, we devise an algorithm which would approximate the counts of itemsets dynamically and discover FIs over a sliding window of a data stream. Our stream mining method, namely the *Sliding-Window based Combinatorial Approximation* (*SWCA*) algorithm, is described as follows.

As mentioned in Section 4, we further divide the sliding window into m equal-size segments of s transactions, and process the sliding of window incrementally in a segment-based manner. The data structure we employ to maintain the summary information is a *lexicographic-ordered prefix tree* modified from the one in [12]. This tree structure mainly maintains I_1, I_2, and FCs of 2-itemsets over the current window of a data stream, also in a segment-based fashion. For each itemset X belonging to I_1 or I_2, the corresponding node in the tree includes a circular array of size m, which corresponds to the m segments of the sliding window, and X's count over the current window is recorded respectively in these m fields. There is also a *pointer* to indicate which field the count of X over S_n is stored. If we combine the counts of all fields in the array, we then obtain the count of X over the current window. Besides, the summary information (i.e., I_1 and I_2) of the current segment S_c is kept separately in an array. In this array, we also maintain I_3 temporarily for the purpose of finding the FC of each 2-itemset at the time when S_c is going to be inserted into the window. When the user invokes a mining request, SWCA then starts approximating the counts of itemsets (whose lengths are above that of the summary) based on the summary, i.e., I_1, I_2, and FCs, stored in the tree.

The SWCA algorithm processes on an on-line transactional data stream. As long as there is no query from the user, SWCA continues receiving and processing the incoming transactions one by one, and handles the sliding of window in a segment-based manner. For each incoming transaction T in the current segment S_c, SWCA enumerates and records the (counts of) first-three-order lengths of subsets contained in T. When S_c is full of transactions, SWCA first finds the FC for each 2-itemset X whose count is not 0 in S_c, and then performs one segment in-out operation. To insert S_c into the window, only I_1, I_2, and the FCs of 2-itemsets are updated into the tree, while the temporarily kept I_3 over S_c is discarded.

When the user invokes a query, SWCA then starts its approximating work. During the process, SWCA first approximates the counts of 3-itemsets dynamically, by choosing different count-values to 1-subsets for different 3-itemsets, according to the relation between the FCs of their corresponding 2-subsets and *ms*. We note that the manner how the FC of a 2-itemset X is found (in our algorithm) is processed according to Definition 2. To summarize, SWCA uses Eq. (1) with k=2 to approximate the counts of 3-itemsets (i.e., m=3). After all 3-itemsets have been approximated, the approximated I_3 is then obtained. In the rest part of approximating work, SWCA uniformly uses Eq. (1) with k=3 to approximate itemsets whose length are above 4.

At the moment we return to the origin. Recall that in Section 3 we have stated and explained that a stream mining algorithm should not work based on an already-known and unchangeable *ms*, which is a constraint our method needs to obey. Now we show that our SWCA algorithm satisfies this constraint adequately. As mentioned before, in the tree of data structure, we maintain no more than I_1 and I_2 (and FCs) over the current window, and the process of mining is proceeding by approximating the counts of itemsets based on this summary information. In SWCA, an itemset is determined as frequent when its approximate count is above *ms*. For different values of *ms* changed by the user, SWCA just applies the approximate equation to calculate (the counts of) itemsets based on the kept summary information, with the dynamically approximating technique employed during the process, and then selects the frequent ones according to respective *ms*. As a result, the usability of SWCA is not affected by a variable *ms*. A user may tune different *ms* at each time, while SWCA is still workable under this circumstance.

Finally we give an analysis about the space complexity of the summary structure. Assume that the source of data stream includes *n* attributes, then the members of I_1, I_2, I_3 are C_1^n, C_2^n, and C_3^n, respectively. Since for each 1-item and 2-itemset, the count is recorded respectively in all *m* segments of the sliding window, and 3-itemsets are kept temporarily only for the current segment, the number of count-values with respect to summary itemsets concerning the preservation is then $mC_1^n + mC_2^n + C_3^n$, which is independent of the number of incoming stream elements and the value of *ms*. No matter how many transactions have been received, and no matter what value of *ms* the user specifies, the consumption of memory of SWCA will be fixed nearly at a certain level.

VI. Experimental Results and Analyses

In this section we are to appraise the proposed algorithm. Four experiments in total have been conducted to evaluate the performance of SWCA. All the experiments were carried out on the platform of personal computer with P4 3.20 GHz dual core CPU and about 800 MB of available physical memory space. The operating system is Windows XP Professional SP2, and the programs of the algorithm are implemented in C++ (and compiled by Dev-C++).

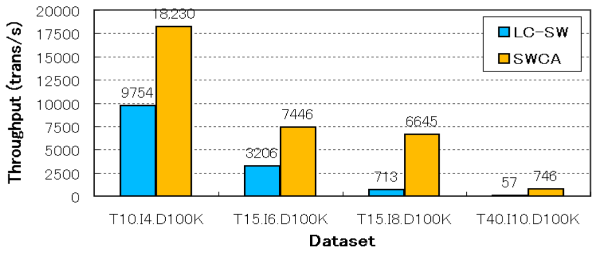
We compare the performance of SWCA with that of the *sliding window* method proposed in [4], which is a variant of the well-known *Lossy Counting* algorithm [3]. We implement the sliding window method in C++ according to the description stated in [4], and denote this method as *LC-SW* in our experiments. We remark that the original method [4] processes the sliding of window transaction by transaction. According to the observation mentioned in [9], this kind of window sliding is much slower and will consume much more memory space than a method with batch-oriented sliding. As a result, we modified from the original method and made the implementation of LC-SW to update the window in a batch of transactions each time.

Table 3 lists the test datasets adopted in our experiments. The first and last ones were downloaded from the website of *FIMI Repository* [14], while others were generated using the *IBM's synthetic data generator* [15]. Every dataset has 1000 different attributes and consists of 100 thousands of transactions. The size of sliding window in our experiments is set to 50,000 transactions for both methods. In LC-SW, each batch receives 10,000 transactions and the window is updated batch by batch. On the other hand, in SWCA, the number of segments (i.e., *m*) is set to 5 and each segment contains 10,000 transactions. Besides, since LC-SW is devised based on the Lossy Counting algorithm, it also has the parameter ε to control the bound of errors. According to the suggestion in [3], we set $\varepsilon = 0.1 \times ms$ for LC-SW.

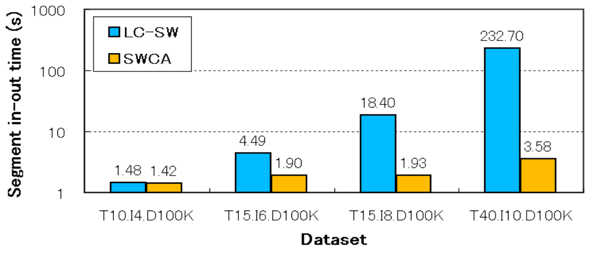
Table 3. Test datasets used in the experiments

Dataset	Transaction length (avg.)	Itemset length (avg.)	Number of attributes
T10.I4.D100K	10	4	1000
T15.I6.D100K	15	6	
T15.I8.D100K	15	8	
T40.I10.D100K	40	10	

The first experiment investigates the efficiency with respect to *throughput* and *average window-sliding time* of both methods. Here throughput is measured as the number of transactions processed per second by the algorithms. We report the experimental result in Fig. 1(a) and (b). In this experiment, the value of *ms* for the first three datasets is set to 0.5%, while for the last dataset *ms* is set to 2% due to its obviously larger values of ***T*** and ***I***. From Fig. 1(a), we found that on all datasets the throughput of SWCA is higher than that of LC-SW, and the difference in time becomes greater as the average length of transactions (***T***) increases. According to Fig. 1(b), the average sliding time of SWCA outperforms LC-SW in all datasets. Besides, it is observed that as the values of ***T*** and ***I*** become larger, both methods will spend more time to complete one (batch/segment-based) window sliding, while the increasing rate of LC-SW is much faster than that of SWCA, which means that SWCA is more efficient.

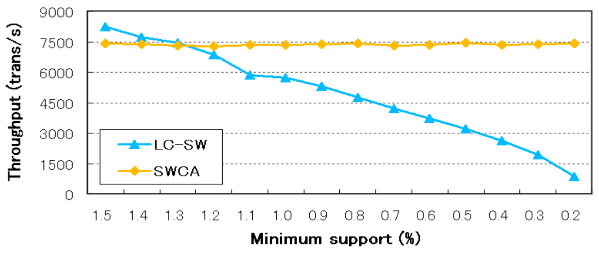


(a) Throughput

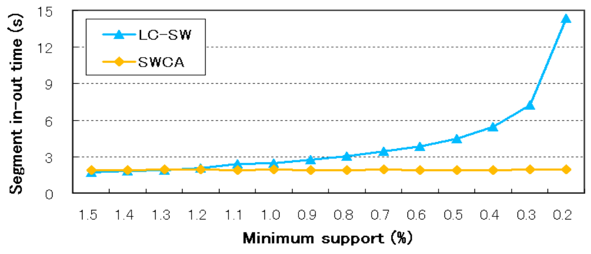


(b) Average window-sliding time

Figure 1. Performance (efficiency) comparison on different datasets



(a) Throughput



(b) Average window-sliding time

Figure 2. Scalability on T15.I6.D100K with varying minimum support threshold

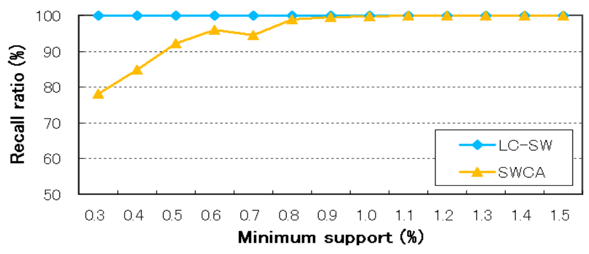
The second experiment evaluates the *scalability* of LC-SW and SWCA with varying the value of *ms*. We measure the throughput and average window-sliding time of both methods, which are similar to those in the previous experiment. The dataset adopted is T15.I6.D100K, and we vary *ms* from 1.5% down to 0.2%. The experimental result is shown in Fig. 2. According to Fig. 2(a) and (b), the performance of LC-SW becomes worse as the value of *ms* decreases. In contrast, the performance of SWCA, with respect to throughput and segment in-out time, is almost independent of the change of *ms*. The scalability of SWCA is well observed through this experiment that it possesses stable performance to both high and low values of *ms*.

The third experiment examines the *accuracy* of both methods. The adopted dataset is T10.I4.D100K and the accuracy is measured as follows. Starting from the point when the sliding window is full of transactions (and is going to slide next), both of the two methods will output a mining result regularly for every sliding. Therefore, the windows of every two successive mining points have 80% of *overlap* with each other. From all the mining results of each value of *ms*, we select several of them to calculate and obtain the average accuracy over the testing data (on that *ms*). The exact sets of FIs are obtained by running an implementation of the *Apriori* algorithm [13] on the snapshots (i.e., the *w* transactions in the current window) at each mining point, respectively. We investigate the accuracy by assessing the *recall ratio* and *precision ratio* of the mining results.

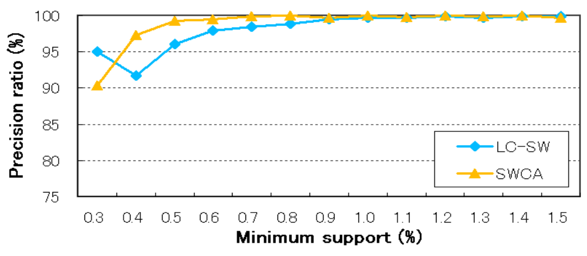
We report the result of this experiment in Fig. 3. From Fig. 3(a) and (b) it is shown that both LC-SW and SWCA achieve high accuracy. In most of the cases, the recall and precision ratios of both methods are above 90% (or even 99%). Even at a pretty low value of *ms* of 0.3%, SWCA still achieves about 80% of recall and 90% of precision, which means that it finds the great majority of FIs over the sliding window. Although on average the accuracy of SWCA is slightly lower than that of LC-SW, by considering the fine efficiency and well scalability of SWCA (as shown in the previous experiments) comprehensively, the performance of SWCA with respect to accuracy is still quite promising.

In the last experiment, we examine the effect of the *dynamically approximating* (DA) technique of SWCA on *accuracy*. The two participants of this experiment are *SWCA-Dynamic* and *SWCA-Static*. The former is the SWCA method with applying the DA technique, while the latter is that without the DA technique and always chooses lower bound of counts (Lbc) to subsets for approximating 3-itemsets. The employed dataset is T15.I8.D100K and the value of *ms* varies from 0.6% to 1.2%. The experimental result is shown in Fig. 4. From Fig. 4(a) it is found that both methods find the whole set of FIs (i.e., are of 100% recall), while the precision ratio of SWCA-Dynamic is higher than that of SWCA-Static for all *ms*, especially at lower values of minimum support. We also calculate the *F-measure* of both methods and present the result in Fig. 4(b), which shows that the F-measure of SWCA-Dynamic (in this experiment) also outperforms that of SWCA-Static.

We remark that the *DSCA* algorithm [12], a data-stream mining method in the landmark model which is based on the approach of *Combinatorial Approximation*, also bounds the count-values of subsets to approximate itemsets. Assume that DSCA is applied to (or transformed into) the sliding window model, it just randomly chooses count-values in the bounded range to subsets since it has no idea to make a decision. If the random strategy is to choose Lbc, then this version of DSCA is just like SWCA-Static. As a result, from the experiment it also indirectly shows that the accuracy of SWCA(-Dynamic) with respect to *precision* outperforms DSCA. The effect of our DA technique on accuracy is adequately proven through this experiment.

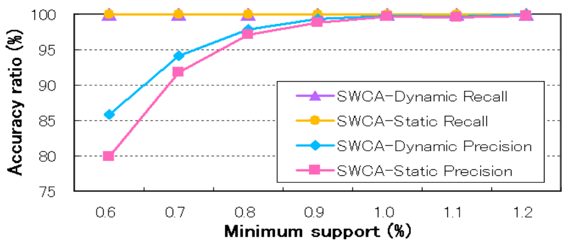


(a) Recall ratio

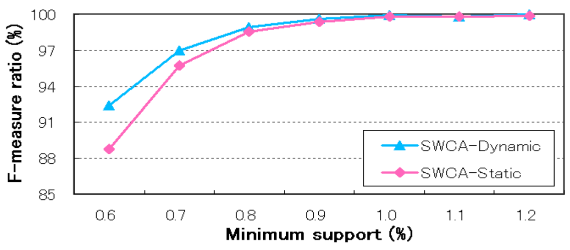


(b) Precision ratio

Figure 3. Accuracy on T10.I4.D100K with varying minimum support threshold



(a) Recall and precision ratio



(b) F-measure ratio

Figure 4. Accuracy on T15.I8.D100K with varying minimum support threshold

VII. Concluding Remarks

In this paper, we study the problem of mining frequent itemsets over the sliding window of a transactional data stream. Based on applying the theory of *Approximate Inclusion–Exclusion*, we devise and propose an algorithm called *SWCA* for finding frequent itemsets through an approximating approach. SWCA conceptually divides the sliding window into *segments* and handles the sliding of window in a segment-based manner. We also introduce the concept of *fair-cutter*, which makes SWCA capable of approximating itemsets dynamically by choosing different parameter-values for different itemsets to be approximated. According to the experimental results, SWCA is quite efficient and possesses good scalability with varying minimum support threshold. Besides, the mining result from SWCA's approximation also achieves high accuracy through the utilization of *dynamically approximating*.

SWCA is a new approach under the sliding window model of data streams. The most obvious difference between SWCA and other existing methods is that SWCA has the *Combinatorial Approximation* as its core. One important feature of SWCA is that its running does not depend on an already-known and constant value of *ms*, which is the case most existing mining methods belong to. As a result, a user would change or tune the value of *ms* each time he/she invokes a query, while SWCA can still work normally and return the mining result. In addition, we devise the novel concept of *fair-cutter*, which is a key contribution of this paper that supports SWCA to approximate dynamically and achieve high accuracy in its mining result.

We remark that the accuracy of SWCA's approximation is possible to be further improved. In this research, the option of choosing count-values to 1-subsets for the dynamically approximating technique is limited to either the upper bound or the lower bound. Nevertheless, other count-values, such as the *average count*, may possibly result in even more accurate approximation for some itemsets. In the future, our works include expanding the dynamically approximating technique, making this technique more flexible, and devising other possible techniques, for the sake of achieving higher and more stable accuracy of mining results.

References

[1] M.N. Garofalakis, J. Gehrke, & R. Rastogi, Querying and mining data streams: you only get one look (A Tutorial), *Proc. 2002 ACM SIGMOD Conf. on Management of Data*, Madison, Wisconsin, 2002, p. 635.

[2] Y. Zhu & D. Shasha, StatStream: statistical monitoring of thousands of data streams in real time, *Proc. 28th Conf. on Very Large Data Bases*, Hong Kong, China, 2002, pp. 358–369.

[3] G.S. Manku & R. Motwani, Approximate frequency counts over data streams, *Proc. 28th Conf. on Very Large Data Bases*, Hong Kong, China, 2002, pp. 346–357.

[4] J.H. Chang & W.S. Lee, A sliding window method for finding recently frequent itemsets over online data streams, *Journal of Information Science and Engineering*, *20*(4), 2004, pp. 753–762.

[5] Y. Chi, H. Wang, P.S. Yu, & R.R. Muntz, Moment: maintaining closed frequent itemsets over a stream sliding window, *Proc. 4th IEEE Conf. on Data Mining*, Brighton, UK, 2004, pp. 59–66.

[6] N. Jiang & L. Gruenwald, CFI-Stream: mining closed frequent itemsets in data streams, *Proc. 12th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, Philadelphia, PA, USA, 2006, pp. 592–597.

[7] C.L. Liu, *Introduction to Combinatorial Mathematics.* McGraw-Hill, New York, 1968.

[8] N. Linial & N. Nisan, Approximate inclusion–exclusion, *Combinatorica*, *10*(4), 1990, pp. 349–365.

[9] J. Cheng, Y. Ke, & W. Ng, Maintaining frequent itemsets over high-speed data streams, *Proc. 10th Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, Singapore, 2006, pp. 462–467.

[10] C.K.-S. Leung & Q.I. Khan, DSTree: a tree structure for the mining of frequent sets from data streams," *Proc. 6th IEEE Conf. on Data Mining*, Hong Kong, China, 2006, pp. 928–932.

[11] B. Mozafari, H. Thakkar, & C. Zaniolo, Verifying and mining frequent patterns from large windows over data streams, *Proc. 24th Conf. on Data Engineering*, Mexico, 2008, pp. 179–188.

[12] K.-F. Jea & C.-W. Li, Discovering frequent itemsets over transactional data streams through an efficient and stable approximate approach, *Expert Systems with Applications*, *36*(10), 2009, pp. 12323–12331.

[13] F. Bodon, A fast APRIORI implementation, *Proc. ICDM Workshop on Frequent Itemset Mining Implementations* (*FIMI'03*), 2003.

[14] Frequent Itemset Mining Implementations Repository (FIMI). Available: http://fimi.cs.helsinki.fi/

[15] Quest Data Mining Synthetic Data Generation Code. Available: http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html