Novel Push-Front Fibonacci Windows Model for Finding Emerging Patterns with Better Completeness and Accuracy

Tubagus Mohammad Akhriza, Yinghua Ma, and Jianhua Li

To find the emerging patterns (EPs) in streaming transaction data, the streaming is first divided into some time windows containing a number of transactions. Itemsets are generated from transactions in each window, and then the emergence of itemsets is evaluated between two windows. In the tilted-time windows model (TTWM), it is assumed that people need support data with finer accuracy from the most recent windows, while accepting coarser accuracy from older windows. Therefore, a limited array's elements are used to maintain all support data in a way that condenses old windows by merging them inside one element. The capacity of elements that accommodates the windows inside is modeled using a particular number sequence. However, in a stream, as new data arrives, the current array updating mechanisms lead to many null elements in the array and cause data incompleteness and inaccuracy problems. Two models derived from TTWM, logarithmic TTWM and Fibonacci windows model, also inherit the same problems. This article proposes a novel push-front Fibonacci windows model as a solution, and experiments are conducted to demonstrate its superiority in finding more EPs compared to other models.

Keywords: Data stream mining, Emerging patterns, Time window models.

Manuscript received Mar. 8, 2017; revised July 3, 2017; accepted July 19, 2017.

Tubagus Mohammad Akhriza (corresponding author, akhriza@stimata.ac.id) is with the Communication and Information System Engineering, Shanghai Jiao Tong University, China and Pradnya Paramita School of Informatics Management and Computer, Malang, Indonesia.

Yinghua Ma (ma-yinghua@sjtu.edu.cn) and Jianhua Li (lijh888@sjtu.edu.cn) are with the School of Information Security Engineering, Shanghai Jiao Tong University, China.

I. Introduction

In the information era, data is being produced and streamed continually in large volumes and at high velocity by a tremendous number of unusual sources such as sensors and social media [1]. Making sense of these large data streams is a task that continues to rely heavily on human judgment. Thus, recognizing patterns contained in the streaming becomes a crucial job [2], [3]. Among all patterns, emerging patterns (EPs) are worthy of mining as they reflect the itemsets' popularity changes from one period to another. The problem of mining EPs has been well studied [4]–[7], and the EP concept has also been broadly implemented to solve problems in many areas such as finding changes in networks [6] and customer behavior in online markets [7].

Streaming data can be divided and loaded into several time windows to find an EP. A window contains several transactions, and itemsets are generated from transactions in each window. An itemset is said to be emerging from two ordered windows W_1 to W_2 if its support growth rate from W_1 to W_2 satisfies a minimum support growth (*mingrowth*) threshold [4]. A mechanism called the tilted-time windows model (TTWM) is proposed to maintain the itemsets' support recorded in all windows [8]. The model assumes that people need support with finer accuracy from recent windows, but they will accept coarser accuracy from older windows [9], [10]. The challenge is condensing support data stored in a large number of windows inside a smaller number of elements of an array, that is, sup[i], in an effective and efficient way.

The maximum number of windows that can be merged into an array's elements, which is called the element's capacity, can be modeled using a certain number sequence. In this paper, as support data are saved in the elements, sup[i] represents the i-th element. Element sup[0] is particularly storing support data from the window that is currently being processed. With a purpose of serving the finest support values, the first few elements' capacities are set to one window. Two models are derived from TTWM. A model called the logarithmic tilted-time windows model (LWin) uses a sequence G = {1; 1[1]; 2[2]; 4[4]; 8[8]; and so on}. Therefore, N windows can be accommodated in a (1 + ^{2}log(N))-elements array [10], while another model [11], called the Fibonacci windows model (FWin), uses a Fibonacci sequence F = {1, 1, 2, 3, 5, and so on} to organize the elements' capacities. Thus, N windows are fit to an n-element array ($N \geq n$) in a relation given by

$$N = \sum_{i=0}^{n-1} X.\mathrm{sup}[i].\mathrm{vol}, \tag{1}$$

where sup[i].vol represents the element's volume, that is, the number of windows being merged inside sup[i], which is not greater than the elements' capacity. A sequence of elements' volumes is called a volume sequence.

The main problem of traditional TTWM, including LWin and FWin, is in their array updating mechanism, which creates null elements even when they should provide the most accurate support. This situation is a significant disadvantage in the process to find EPs, which checks the support growth between elements sup[0] and sup[j], $j > 0$. As a consequence of such data incompleteness, some EPs at certain time windows cannot be found.

This article first introduces a novel model called the push-front tilted-time windows model (PF-TTWM) as the solution for the data incompleteness of TTWM. A push-front mechanism is applied to eliminate nulls, and first elements can provide support with the finest accuracy. Second, by applying this to a Fibonacci window, a novel push-front Fibonacci window model (PF-FWin) is proposed. Fibonacci sequence F is used to organize an element's capacity so that N windows can always be accommodated in an n-element array ($N \geq n$) by the relation given in (1). X.sup[i].vol $\in$ {1, 2, . . . , F[i]} in PF-FWin, while in FWin, X.sup[i].vol $\in$ {0, F[i]}.

In experiments, the performance of PF-FWin is compared to those of FWin and LWin. Two datasets are used: a synthetic dataset containing 1M records generated using the IBM Quest data generator, and a real online retail dataset containing 541,000 records from the UCI repository. The process of accommodating N windows of streaming data into an n-element array is implemented using a deterministic finite automaton. Transactions are processed by an online streaming method. The emergence of X is evaluated by inspecting its support stored in X.sup[0] and another X.sup[j], $j > 0$. X is mined out as an EP if X's support and support growth meet the given thresholds. With regard to the results, the number of EPs found using PF-FWin was up to 90% and 275% higher than when using LWin and FWin, respectively, in the first experiment. In the second experiment, PF-FWin processed more transactions than the two other models.

The rest of article is organized as follows. Section II contains a review of the literature related to our work. Section III defines the proposed solution. Section IV discusses experimental work and its results, and Section V presents the conclusion of this work.

II. Related Works

This work contributes to the area of research about finding EPs in data streams using TTWM, which is still very rarely found in the literature. On the other hand, LWin was recorded as the first model that maintained the itemsets and their support using the TTWM approach, although it was not designed to find EPs [8]. FWin is known as the second method that adopted TTWM. In this section, the basic concept of EP mining is explained, followed by details of TTWM, LWin, and FWin, particularly in the array updating mechanism.

1. Mining the Emerging Patterns

Mining EPs has been well studied since 1999 [5]–[7], and the implementations of EPs in many areas are found in the literature as well. For example, implementations in the medical area can distinguish similar diseases [12], classify cancer diagnosis data [13], and profile leukemia patients [14]. In the networking environment, EPs are implemented to find significant differences in network data streams [6], or to detect masqueraders [15].

EP is a class of frequent patterns (FPs) where the itemsets are generated using association rule mining. An itemset X is said to be frequent in dataset D if X's support in D, written as $\mathrm{sup}_D(X) \geq minsupp$, a user-given minimum support threshold [16]. In other words, an EP contains related items because their appearance together in D is frequent with respect to *minsupp*.

An EP is formally defined as follows [5]. Given an ordered pair of datasets D_1 and D_2, set I contains all items in both datasets, and $X \subseteq I$ is an itemset. Let $\sup_i(X)$ denote the support of X in D_i. The support growth rate of X from D_1 to D_2, denoted as $GR(X)$, is defined as

$$GR(X) = \begin{cases} 0 & \text{if } \sup_1(X) = 0 \text{ and } \sup_2(X) = 0, \\ \infty & \text{if } \sup_1(X) = 0 \text{ and } \sup_2(X) \neq 0, \\ \frac{\sup_2(X)}{\sup_1(X)} & \text{otherwise.} \end{cases} \quad (2)$$

Given that $\rho > 1$ is a *mingrowth*, an itemset X is said to be a ρ-emerging pattern (or simplify ρ-EP or EP) from D_1 to D_2 if $GR(X) \geq \rho$. The EP mining problem is, for a given ρ, to find all ρ-EPs. In a special case when $\sup_1(X) = 0$ and $GR(X) \geq \rho$, X is called a jumping EP (JEP).

The EP mining concept was initially applied to two static datasets of D_1 and D_2. The data in both datasets come from the same source, but are collected at two different time stamps, for example, the transactional data of a supermarket recorded in different months. Alternatively, D_1 and D_2 can also come from different sources, for example, transactional data from different supermarkets. This notion was adapted to two classes of dynamic datasets accordingly, such as in the data stream environment, and relevant examples involve finding the changes in the network by inspecting the IP traffic patterns captured from network routers [6], and to approximate the EP from two different classes of streaming data [7].

EPs can also be found in one class of streaming or temporal dataset. In this case, the streaming is divided into several time windows. Each window contains a number of transactions. An EP is evaluated between windows W_1 and W_2, where W_1 is the most recent window. A method named Dual Support Apriori for Temporal Data is proposed to find the emerging trends from temporal datasets using a sliding windows model [4]. The sliding windows model is also used to find contrast patterns in imbalanced classification [17], to find emerging melody patterns [18], and to discover contrast patterns in changing data [19]. However, the problem of long-term evaluation of EPs at multiple time windows is not covered in this sliding windows model because this model only concerns two windows, the current and previous windows, while the older windows are ignored.

2. Tilted-Time Windows Model

The TTWM was proposed to store and maintain an itemset's supports from a certain number of windows at

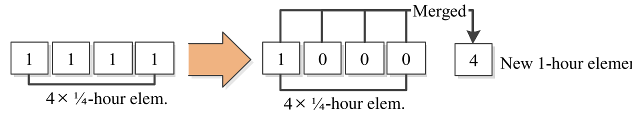

Fig. 1. Element updating in tilted-time windows model.

multiple time windows in the long term. As explained in [8]–[10], people actually need the most accurate supports only from some recent windows, while coarser accuracy is acceptable from older windows. Therefore, support data in a large number of windows should be condensed efficiently into a smaller number of array elements.

Technically, each itemset X stores its support data in an n-element array, denoted as X.sup[i], or sup[i] to simplify. Each sup[i] has a capacity, that is, the maximum number of windows that can be merged inside a respective element, while sup[0] stores support obtained from the window that is currently being processed. TTWM models the capacities of sup[i] into a particular number sequence, that is, a capacity sequence. The smaller the capacity, the finer the accuracy of support. The capacities of the first few elements are set to 1 to ensure the finest support accuracy. In most models, the capacities of elements of sup[i] $\leq$ the capacities of sup[j], $i < j$.

Suppose the transactions are collected and then processed in every 1/4 h, but the manager requires only the most accurate supports in the last 1-h processes. Thus, four elements, that is, sup[0,0,0,0], are initially created to accommodate support from four windows of 1/4-h data processing. Figure 1 demonstrates TTWM when the elements are updated. After all 1/4-h elements are occupied, or X.sup[1,1,1,1], as support found in current window will be put in sup[0], supports in 1/4-h elements first are merged and stored into the 1-h element. Next, all 1/4-h elements are nullified, and sup[0] is now ready to store the current support, or the array becomes sup[1,0,0,0,4] (Fig. 1). This approach facilitates 1-day data processing only by 4 (1/4-h element) + 24 (1-h elements) = 28 elements, instead of using 4 × 24 = 96 elements to store all 1/4-h processing results for 24 h. It is an efficient solution [7]–[9].

3. Logarithmic Tilted-Time Windows Model

A model derived from TTWM, named logarithmic TTWM (also called LWin), was first introduced in [10] with the goal of storing and maintaining supports of itemsets for the long term. The decision-maker benefited from this approach since a time-related query about the pattern's popularity trend can be evaluated, such as patterns that are popular once, seasonally popular, or forever popular. Such a benefit cannot be harvested from the other data windows models [9], [10], such as sliding windows [4], [17]–[21], damped windows [22], [23], or landmark windows models [24], [25].

LWin is designed to deal with storing the results of FP mining in an offline data stream. The streaming transactions are divided into N windows containing a uniform number of transactions. The logarithmic approach is proposed to reduce the number of elements more significantly, compared to the original TTWM. N windows can be accommodated in only $\{1 + {}^{2}\log(N)\}$ elements. For instance, 1,024 windows are fit in only 11 elements, which is very memory efficient.

LWin uses a sequence $G = \{1\}\cup\{1[1], 2[2], 4[4], 8[8],$ and so on$\}$ to organize the capacity of sup[i] elements, $i \geq 0$. The sequence is a geometric sequence with ratio $r = 2$. Except each element (called the main element), in maintaining the entire structure there are intermediate elements that normally are null, named sub[i], and have the same capacity as the main element, that is, sup[i] [7]–[9]. Those articles also explain that the sequence's ratio is a positive integer and is changeable in accordance with the analyst's needs, although they only explain how LWin works using $r = 2$. We can see three 1s in sequence G, which respectively represent the most accurate support stored in sup[0], sup[1], and sub[1].

Given a $2n$-element array, these are sup[i] and sub[i], $i = 0$ to $n - 1$. The updating mechanism of the array elements in LWin applies the following procedures:

1. sup[0] is always filled with the support in the current window.
2. sup[1] is filled only by shifting sup[0] into it.
3. sub[i] is filled only by shifting sup[i] into it, while sup[i], $i > 1$, is filled by sup[i] = sup[$i - 1$] + sub [$i - 1$], and sub[$i - 1$] is nullified afterward.
4. The first updating target is a sub[i] = 0, where i is the lowest index, and the updating continues to its main element sup[i], followed by updating all sup[j] and sub[j], $j < i$, using procedure numbers (1), (2), and (3).
5. When all sup[i] and sub[i] are occupied, increment $n = n + 1$, and create a new pair of elements sup[n] and sub[n] at the back of the array. sup[n] becomes the first updating target, followed by all sup[i] and sub[i], $i < n$, using procedures (1), (2), and (3).

However, LWin must overcome two situations in its element updating process. If N is exactly 2^n, then N windows are fit to the (n + 1)-main-element array. For example, when $N = 8$ and thus $n = 3$, 8 windows are exactly accommodated in 4 elements, where the volume sequence is like {1, 1[], 2[], 4[]}; [] represents an empty sub[i]. However, if N is not exactly 2^n, then N windows will be accommodated by the (n + 1)-main-element array, plus some intermediate element(s). For example, when $N = 9$ and thus $n \approx 3$, 9 windows will be condensed into a four-element array, plus an intermediate element where the formed volume sequence is like {1, 1[1], 2, 4}.

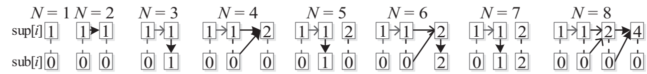

Fig. 2. Volume sequence and updating steps in LWin.

An illustration of the volume sequence updating when X is found in $N = 1$ to 8 windows is given in Fig. 2. The dashed line, bold arrow, and thin arrow represent a pair of sup[i] – sub[i], the first step, and the next steps, respectively. From this illustration, it is known that the number of elements, hence the memory needed by each itemset, is actually a maximum of $2 \times \{1 + {}^{2}\log(N)\}$.

4. Fibonacci Windows Model

FWin or the Fibonacci windows model was introduced to improve the memory efficiency issue that occurred during LWin usage, and also to deal with online data stream mining [14]. The main difference between FWin and LWin is that instead of using a "doubled" geometric sequence, FWin uses a Fibonacci sequence F = {1, 1, 2, 3, 5, and so on} to organize the elements' capacity. Thus, no intermediate element is needed in the updating process. In addition, the distance between capacities in F[i] is smaller than the distance in G[i], $i > 2$. According to the principle of TTWM, FWin provides better accuracy than LWin.

Given an n-element array, sup[i], $i = 0$ to $n - 1$, the element updating procedure in FWin is described as follows:

1. sup[0] is used to accommodate support in the current window
2. sup[1] is updated by shifting sup[0] into it.
3. sup[i], $i > 1$, is updated by merging sup[$i - 1$] and sup[$i - 2$] into it, or sup[i] = sup[$i - 1$] + sup[$i - 2$]. Then, sup[$i - 1$] is nullified afterward. By decrementing $i = i - 2$, procedure (3) is repeated.
4. The first updating target is a sup[i] = 0, where i is the lowest index, and the updating is continued to all sup[j], $j < i$ using procedure numbers (1), (2), and (3).
5. When there is no sup[i] = 0 left, increment $n = n + 1$, so a new element is created at the back of the array, sup[n]. Since sup[n] = 0, it becomes the first updating target by using procedure (4).

An illustration of the volume sequence updating when X is found in $N = 1$ to 7 windows is given in Fig. 3. The

https://doi.org/10.4218/etrij.18.0117.0175

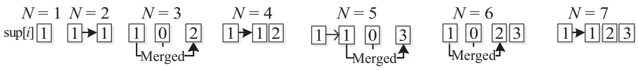

Fig. 3. Volume sequence and updating steps in FWin.

bold arrow and thin arrow represent the first and next steps, respectively.

Using FWin, N windows can be accommodated in an n-element array, $N \geq n$, such as (1). The value of sup[i].vol is either 0 or $F[i]$. For example, a 15-element array accommodates 1,596 windows. If one window is equal to 1 day of data processing, a 15-element array is efficiently fit for 4.37 years of data processing.

However, both LWin and FWin still possess some shortcomings, particularly with regard to data incompleteness. Elements sup[1] or sub[1], which should provide the most accurate support, may even be empty. Consequently, queries about support values at several past timestamps cannot be answered. The EPs in those timestamps also cannot be evaluated; hence also a change in the itemset's popularity. The solution to these issues is offered in this work via the proposal for a novel Push-Front Tilted-Time Windows Model and its derivation model, that is, the Push-Front Fibonacci Windows Model. These are explained in the next section.

III. Proposed Methodology

This section explains the details of PF-TTWM, followed by the proposed PF-FWin and an automaton for PF-FWin as an updating mechanism. The EP mining criteria, which are used to find EPs using PF-FWin, are explained afterward.

1. Push-Front Tilted-Time Windows Model

An illustration of the proposed PF-FWin model is shown in Fig. 4. The main difference between PF-TTWM and TTWM is that when all elements reach their respective capacities, PF-TTWM inserts a new element at the front of the array, so there is no null element created. In a case given in Subsection II-2, the push-front operation makes X.sup = [1,1,1,1] become X.sup = [1,1,1,1,1], instead of X.sup[1,0,0,0,4], such as that produced by TTWM. Support in sup[3] becomes sup[4] and represents a 1-h element, while a new sup[0] will be filled with new data just arrived.

The main goals of tilting the time windows in PF-TTWM are derived from TTWM (that is, to reduce the memory used to store the supports) and to eliminate null elements in PF-TTWM. In the new proposed model, many front elements are filled with one window. Thus, PF-TTWM can provide better data completeness and accuracy, compared to traditional TTWM.

Programmatically, the push-front operation is implementable directly in C++ language using the *pushfront*() method. It is a member function of the deque (double-ended queue) container. The amortized time complexity of *pushfront*() is $O(1)$, which is very efficient because element insertion is done at the beginning of the data structure.

2. Push-Front Fibonacci Windows Model

PF-TTWM can improve FWin's performance to find EPs. Using PF-FWin, N windows, in which X is found, can always be accommodated into an n-element array ($N \leq n$) by referring to (1), where $X.\text{sup}[i].\text{vol} \in \{1, 2, \ldots, F[i]\}$.

Given an n-element array sup[i], i = 0 to n − 1, the complete updating procedures are as follows:

1. For $i = n - 1$ to 0, do /*Here, the last element, that is, sup[n − 1] becomes the first updating target */
   a) sup[i] + = sup[i − 1]
   b) Shift sup[i − 1] ← sup[i − 2], . . . , ← sup[1] ← sup[0]
   c) Particularly, if i = 0, then sup[0] ← support in the current window.
2. If the volume of sup[n − 1] reaches its capacity, then $n = n - 1$, and repeat procedure (1). This step means that after sup[n − 1].vol reaches its capacity, then the next updating target is sup[n − 2], and so on.
3. If all sup[i].vol reach their capacity, that is, the volume sequence = the Fibonacci sequence, a new element is pushed to the front of the array, sup[0], while the current array's indices are all incremented by 1 accordingly.

Figure 5 shows an illustration of the updating mechanism and the change in the volume sequence of PF-FWin. The bold arrow and thin arrow represent the first and next steps, respectively, where the *pf* arrow represents

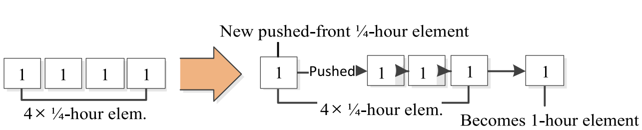

Fig. 4. Proposed push-front tilted-time windows model.

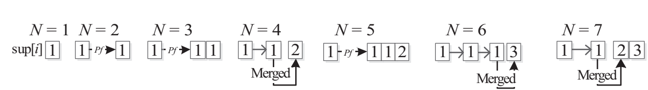

Fig. 5. Volume sequence and updating steps in PF-FWin.

the push-front operation, and the merged-arrow shows the merging process sup[i] + = sup[$i-1$], respectively.

3. PF-FWin Updating Automaton

PF-FWin's updating procedures above are modeled in a deterministic finite automaton called the PFwin Automaton (PFwinA). Formally, the automaton is defined as PFwinA(S, I, S_0, Z, f), and the tuples are explained as follows:

1. State set S = {**Uc**: update current element, that is, sup[m], $m = n - 1$; **Up**: update previous element, that is, sup[$m-1$]; **Pf**: push front the array}
2. Input set I = {isFiboSeq, isNotFiboSeq, isFiboElement, isNotFiboElement},
3. A start state $S_0 = \{Pf\}$
4. Stop states set $Z = \{Uc, Up, Pf\}$
5. Next-state function f: $S \times I \rightarrow S$, a function to direct the current state to the next state, after reading the input. The transition table of function f is given in Table 1, while the transition graph is given in Fig. 6. Additionally, Table 1 also lists the steps taken after a current state reads an input.

PFwinA describes four conditions that may appear during the updating process, and they are defined as the input of the automaton. The first condition is called *isFiboSeq*. It means the volume sequence generates a Fibonacci sequence or Σsup[i].vol = $\Sigma F[i]$. The second condition is *isNotFiboSeq*, which is true if sup[$n-1$].vol = $F[n-1]$ but Σsup[i].vol $\neq \Sigma F[i]$. The third condition is *isFiboElement*, which is reached when sup[$n-1$].vol = $F[n-1]$. The last condition is *isNotFiboElement* to explain sup[$n-1$].vol < $F[n-1]$. If a condition is met after the current state is reached, the step given in the last column of Table 1 is applied. The current element being updated is always sup[m], where $m = n - 1$. When a previous element should be updated, m is decremented by one, $m = m - 1$, and sup[m] is updated. The current element is now reset to sup[m].

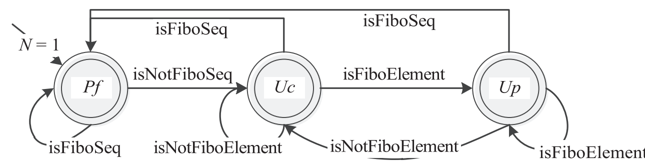

Fig. 6. PFWin automaton graph transition.

The iteration is started by pushing the first element to the front of the array, and thus PFwinA is started from the Pf state, where $N = 1$, or sup[$n-1$].vol = 1. The automaton can stop at any state, and then it is waiting for the next new window addition.

To prove that PFwinA can generate a volume sequence as defined in (1) using the push-front approach, the following properties are developed and their proofs are given:

Property 1: When a Fibonacci sequence has not been developed yet, PFwinA repeatedly updates sup[m] until sup[m].vol = $F[m]$. We have to prove that for $\forall s \in S - \{Pf\}$, $f(s$, isNotFiboElement$) = t \Rightarrow t = Uc$. Pf is excluded since it represents a state where isFiboSeq is true.

Proof: By contradiction, assume $\exists s \in S - \{Pf\}$ where $f(s$, isNotFiboElement$) = t \wedge t \neq Uc$. By inserting all $s \in S - \{Pf\}$ into the next-state function above, $f(\{Up, Uc\}$, isNotFiboElement$) = f(Up$, isNotFiboElement$) \cup f(Uc$, isNotFiboElement$) = \{Uc\}$. Since all function results are $t = Uc$, thus the assumption is not correct. Therefore, Property 1 is correct. ■

Property 2: After sup[$n-1$].vol = $F[n-1]$, PFwinA continues to update sup[$n-2$]. The proof is clearly given in Table 1 that $\forall s \in S$, $f(s$, isFiboElement$) = t \Rightarrow t = Up$, which means PFwinA is updating the previous element,

Table 1. State transition table for PFwin automaton.

No	Current state	Input	Next state	Step taken
1	*Pf*	isFiboSeq	*Pf*	Push-front a new element sup[0]
2	*Pf*	isNotFiboSeq	*Uc*	Update current element sup[m]
3	*Uc*	isFiboSeq	*Pf*	Push-front a new element sup[0];
4	*Uc*	isFiboElement	*Up*	Update previous element: $m = m - 1$; update sup[m]
5	*Uc*	isNotFiboElement	*Uc*	Update current element sup[m]
6	*Up*	isFiboSeq	*Pf*	Push-front a new element sup[0]
7	*Up*	isFiboElement	*Up*	Update previous element: $m = m - 1$; update sup[m]
8	*Up*	isNotFiboElement	*Uc*	Update current element sup[m]

https://doi.org/10.4218/etrij.18.0117.0175

that is, by decrementing $m = m - 1$ and then updating sup[m].

Property 3: The second push-front operation taken after {sup[i].vol} = a Fibonacci sequence generating sup[n] = $F[n]$, $n > 1$, $i = 0$ to $n - 1$.

Proof: Suppose the volume sequence = a Fibonacci sequence. Thus, {sup[i].vol} = $\{1, 1, 2, \ldots, F[n-2], F[n-1]\}$. The first push-front makes volume sequence = $\{1\} \cup \{1, 1, 2, \ldots, F[n-2], F[n-1]\}$. The summation of $F[n-1] + F[n-2]$ creates the next Fibonacci element $F[n]$, and is stored in sup[n]. ■

4. Finding EPs Using PF-FWin

Finding EPs over the data stream using PF-FWin (and other TTWM models) requires dealing with large supports as the merging results of some supports. As a consequence, $GR(X)$ formulation and *mingrowth* become two main keys that affect to the EP quantity that can be found from the streaming.

This article evaluates an emergency of X by inspecting X.sup[0], that is, the current support and another X.sup[j], $j > 0$. X is said to be a ρ-EP, $\rho > 0$, if the two following criteria are met:

1. X is frequent in the current window, or X.sup[0] ≥ *minsupp*.
2. $GR(X)$ from sup[j] to sup[0] ≥ ρ, where $GR(X)$ is defined in this work as

$$GR(X) = \frac{(X.\text{sup}[0] - X.\text{sup}[j])}{X.\text{sup}[j]}. \tag{3}$$

Such a $GR(X)$ formula is slightly different from one defined in the original work (introduced in Section II) because sup[j], $j > 1$, possibly contains a large support value. Thus, X's support growth is said to be significant if it reaches a certain percentage, for example, 20%.

IV. Experimental Works, Results, and Discussions

Two experiments are conducted to evaluate PF-FWin performance compared with LWin and FWin to find EPs in online transaction data streaming. The EP quantity as well as the support accuracy and data completeness are the evaluated performance indicators. The online streaming process means that itemsets are directly generated from each transaction as it arrives. Each generated itemset will also be directly evaluated as to whether it is an EP using criteria explained in Subsection III-4. The runtime performance is also evaluated.

Programs for the three models are developed using C++ and compiled in Visual Studio Express 2013. Those programs are individually run on three different PCs with the same specifications, that is, an Intel Core i3 CPU @ 3.30 Ghz, 2-GB RAM, under Windows 7 64-bit. A container in C++'s Standard Template Library, that is deque, is used to store the supports in all models. A member function of deque, that is, pushfront(), is used to run the push-front operation in PF-FWin's element updating process.

1. Experimental Design

In the first experiment, a synthetic transaction dataset generated using the IBM Pro Quest Data Generator is used. It contains 1 million transactions and 243 items, where one transaction has 3–10 items. Each window contains a fixed number (10K) of transactions. Thus, there are a total of 100 windows. *Minsupp* is set to 100, while *mingrowth* ρ = 20%. Higher *mingrowth* values, for example, 50%, have also been experimented on, but since the number of found EPs as not large, this will not be discussed. The same dataset and thresholds are also used in another case in which a new window is created in every 10 min of transaction processing.

The second experiment uses the Online Retail Dataset, which is available in the UCI repository [26]. The dataset contains 3,958 items and about 541,000 records of invoices (transactions) in which each invoice lists a number of singleton items. As some invoices have hundreds of items while others only have a very few, in order to make the number of items even, large invoices are divided into smaller ones. Thus, each invoice contains 10 items maximally. This approach was discussed in [24], [25], where the streaming was divided into some item buckets (that is, windows) in order to keep the program running under limited memory resources. A new window is created after 2,500 invoices are processed in a previous window. *Minsup* and *mingrowth* are set to 50% and 20%, respectively.

In order to make the data easier to compare, only itemsets with ≤ 3 items are processed, considering that longer itemsets usually have smaller support and short itemsets have a higher probability to emerge [16]. In addition, since we use an online data stream, memory resources are crucial features to consider, especially in a single-machine environment.

2. Results and Discussions

A. First Experiment Results

Discussion of the first experiment results starts with the number of EPs found by the three models, as charted in

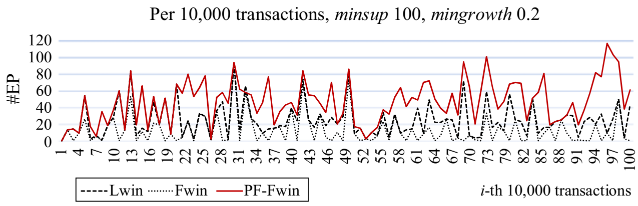

Fig. 7. Number of EPs found by LWin, FWin, and PF-FWin per 10K transactions.

Fig. 7. In total there are 4,708 EPs found by PF-FWin, while FWin and LWin only found 1,255 and 2,469 EPs, respectively. This means that the performance of PF-FWin to find EPs is about 1.9 times and 3.75 times (or about 90% and 275%) higher than LWin and FWin, respectively.

The runtime performance does not show significant differences between the three methods, whereas PF-FWin applies a quite different approach compared to the traditional ones. PF-FWin finished processing all 1M transactions by creating 100 windows in 11H(our):27(M)inutes, while LWin and FWin, respectively, spent 11H:48M and 11H:28M. Two methods of FWin are faster than LWin. In total, 814,895 different itemsets in 100 windows, or about 814.959 itemsets/window/10K transactions, are generated.

Compared to the two other models, LWin took the longest time to finish all processes because it uses a "double" geometric sequence to organize the array's elements. Thus, the number of elements that must be checked doubles as well. Our observation concluded that the time-complexity worst case of LWin is when all or almost all sup[i] and sub[i] are occupied, or N is almost reaching $2n$, such as when $N = 15$. The volume sequence for $N = 15$ using LWin is {1, 1[1], 2[2], 4[4]}, and as shown, all sup[i] and sub[i], $i > 0$, are filled. Thus, when an EP evaluation is performed, the LWin algorithm must double-check all elements.

As a comparison, the volume sequences of FWin and PF-FWin when $N = 15$ are {1, 1, 2, 3, 0, 8} and {1, 1, 1, 1, 3, 8}, respectively, but the number of EPs found by LWin, FWin, and PF-FWin when $N = 15$ is 15, 11, and 66 EPs, respectively. The number of elements in an array of all models is actually quite similar. The reason PF-FWin produces more EPs than the other methods is because of the presence of many 1s in front of the array. These instances show that the better support accuracy of PF-FWin improved the number of EPs that can be found by this model. The worst case of PF-FWIn occurred when all sup[i].vol reached their capacity, or when the volume sequence = a Fibonacci sequence because the array contained only two 1s in its front elements.

Figure 7 shows that LWin outperforms PF-FWin in two windows, that is, $N = 7$ and 33. If elaborated, PF-FWin's and LWin's volume sequences in $N = 7$ are {1, 1, 2, 3} and {1, 1[1], 2, 4}, respectively; while for $N = 33$ they are {1, 1, 2, 3, 5, 8, 13} and {1, 1[1], 2, 4, 8, 16} for their respective models. The number of EPs found by these models when $N = 7$ is 5 and 6 EPs, respectively, and when $N = 13$, the results are 58 and 66 EPs, respectively. All sup[i].vol in PF-FWin reached their capacity. Thus, the array has only two 1s in its first elements. By contrast, the array in LWin has three 1s and makes more itemsets for the *mingrowth*. Nevertheless, these instances are effective proofs that having more 1s in an array opens an opportunity for the algorithm to find more EPs.

The best case for PF-FWin is reached after the volume sequence = Fibonacci sequence and push-front operations are performed thrice afterward. The reason for this situation is because many 1s are generated in front of such a sequence. After the volume sequence = Fibonacci sequence, the first push-front operation creates a new element in front of the array. The second push-front merges sup[$n - 2$] into sup[$n - 1$], and thus sup[$n - 1$] = F[$n - 1$]. The third push-front updates sup[$n - 2$] + = sup[$n - 3$]. As a result, many 1s are then pushed into the front of the sequence, and accordingly the support accuracy is improved and also becomes the best in this state. An example for this explanation occurs when comparing volume sequences when $N = 12$ and $N = 15$.

By contrast, results from additional observations also quite similar with previous results (Fig. 8). The number of EPs found by PF-FWin, FWin, and LWin, respectively, is 3,523, 686, and 1,962 EPs. In this case, PF-FWin also outperforms the other models up to 5.13 and 1.79 times, respectively.

In the first experimental approach, as the number of transactions in all windows is uniform, the number of itemsets created and number of EPs found in each window are also constant. However, these numbers are not the same for each window in an additional experiment. This phenomenon likely occurred because of two reasons. First, although the timer was set to 10 min (or other durations), the computer results for the time calculation have a

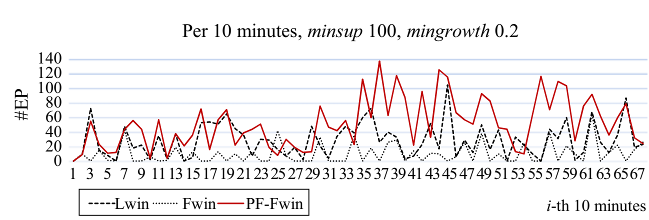

Fig. 8. Number of EPs found by LWin, FWin, and PF-FWin per 10 min.

https://doi.org/10.4218/etrij.18.0117.0175

difference of milliseconds. This difference affects the number of itemsets generated in each window. Hence, there is the opportunity for the itemset to be frequent and then emerge in later windows. This also becomes the reason that for certain windows, the number of EPs found by FWin and LWin is larger than those found by PF-FWin.

Almost all windows created by both experiment approaches. However, PF-FWin always outperforms the other two models to find EPs. If we review the *GR* formula, then an itemset X can become an EP only if X.sup[j], hence X.sup[j].vol, is not that large. Among the three models, only PF-FWin can meet such a requirement because in certain circumstances, many sup[i].vol contain only one window. Therefore, even if ρ is made larger than 0.2, such as 0.5, PF-FWin still becomes the winner.

By contrast, FWin produces the lowest number of EPs because several elements are empty in some situations. For example, when N = 14, the volume sequences of PF-FWin, FWin, and LWin, respectively, are {1, 1, 1, 1, 2, 8}, {1, 0, 2, 3, 0, 8}, and {1, 1, 2[2], 4[4]}. While LWin even has two 1s in front of the array, FWin contains only one 1; its sup[1].vol is even empty, whereas it should provide the most accurate support. Accordingly, very few itemsets have the opportunity to be frequent and hence emerge in FWin.

B. Second Experiment Results

In addition to the quantity of EPs, the second experiment also addresses EP accuracy produced by the three methods.

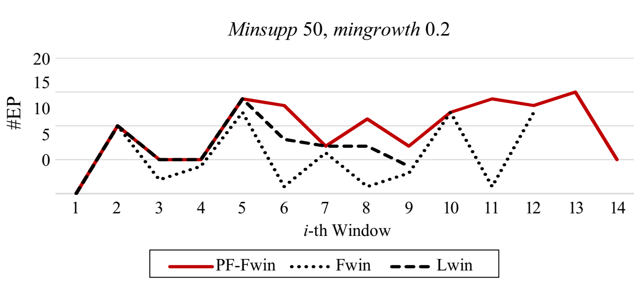

Fig. 9. Number of EPs found by three methods in *i*-th window.

Figure 9 shows the number of EPs produced by the compared methods, where the superiority of PF-FWin in discovering more EPs is clearly shown. PF-FWin, FWin, and LWin discover, respectively, 131, 64, and 60 EPs, which means that PF-FWin found about two times more EPs than the other models. The reasoning for this achievement was discussed in the previous subsection. Moreover, PF-FWin processes more invoices (in 14 windows) than the other methods, that is, FWin (12 windows) and LWin (9 windows). FWin and LWin failed owing to a lack of memory during the array's updating process. This result shows a higher memory-efficiency provided by PF-FWin compared to the others.

In the 8th window, for example, PF-FWin found 11 EPs, while FWin and LWin found only 1 and 7 EPs, respectively. Three of 11 itemsets that emerge in the 8th window are listed in Table 2. The supports' arrays, which store the respective itemset supports and the array's volume sequences, are also given. The sequence shows the number of windows being merged into the elements. Hence, it indicates the accuracy of supports stored in respective elements.

PF-FWin found all three EPs, but only EP no. 2 {22084} can be found by FWin. By contrast, LWin found itemset{85099C} as an EP, but it could not be found by FWin. Itemset{85123A}, which is registered as a product called *White hanging heart t-light holder*, is found to be emerging by PF-FWin because its support growth ≥ 20% from sup[2] (75) to sup[0] (95). Using FWin and LWin, this itemset is not found to be emerging because the support growth threshold is not satisfied by sup[0] and the other elements. LWin found itemset{85099C} to be emerging from sup[1] (34) to sup[0] (51) with a support growth of 50%, but FWin could not find it as an EP because of the same reason explained previously.

With regard to accuracy, PF-FWin is able to provide supports with 100% accuracy in three elements, that is, from sup[0] to sup[2], since only one window is contained by such elements. LWin can still provide 100% accuracy in the current window (sup[0]) and the previous window

Table 2. Some EPs found in 8th window.

No	EP	[Support's arrays] [Array's volume sequence]		
		PF-FWin	FWin	Lwin
1	85123A	[95,91,75,189,287] [1,1,1,2,3]	[95,0,166,0,476] [1,0,2,0,5]	[95,91,181,370] [1,1[0],2[0],4[0]]
2	22084	[69,48,45,28,29] [1,1,1,2,3]	[69,0,93,0,57] [1,0,2,0,5]	[69,48,61,41] [1,1[0],2[0],4[0]]
3	85099C	[51,34,37,77,73] [1,1,1,2,3]	[51,0,71,0,150] [1,0,2,0,5]	[51,34,82,105] [1,1[0],2[0],4[0]]

(sup[1]); however, FWin can only provide the finest accuracy in the current window. By contrast, PF-FWin also can provide better data completeness than the other methods because there no null element exists in its support's array. The other two methods contain null element(s). In FWin, for example, if the support information in a previous window is requested, then such a request cannot be answered because sup[1] is empty, whereas it should provide support with the finest accuracy. Alternatively, support stored in sup[2] can be used as an approximation for sup[1], although it is not 100% accurate since the support in sup[2] is a merging of supports stored in the two past windows.

In the real online business world, the marketing manager often needs to make a critical decision instantly, such as the itemset(s) that should be supplied into a marketplace based on emerging demand. Experimental works show that PF-FWin is the best solution for this problem because it is more time and memory efficient in maintaining found itemsets and their support in the long term, compared to the other models.

V. Conclusion

The proposed PF-TTWM approach offers better data completeness and accuracy compared to traditional TTWM. In this implementation, PF-FWin also shows superior time and memory efficiency to maintain the found itemsets and their support for a long term. As demonstrated in experiments, data completeness and accuracy are two quality characteristics that must be provided by tilted time-window model approaches, since these characteristics improve the EP quantity that can be found by the models.

Acknowledgements

This research work was funded by the National Key Research and Development Project of China (2016YFB0801003), the Key Laboratory for Shanghai Integrated Information Security Management Technology Research, and the State Grid Corporation of China (SGCC) Science and Technology Project (SGRIXTKJ [2017]133).

References

[1] B. Wixom et al., "The Current State of Business Intelligence Inacademia: The Arrival of Big Data," *Commun. Assoc. Inform. Syst.*, vol. 34, Jan. 2014, pp. 1–13.

[2] A. Balliu, D. Olivetti, O. Babaoglu, M. Marzolla, and A. Sîrbu, "A Big Data Analyzer for Large Trace Logs," *Computing*, vol. 98, no. 12, Dec. 2016, pp. 1225–1249.

[3] A. Gandomi and A. Haider, "Beyond the Hype: Big Data Concepts, Methods, and Analytics," *Int. J. Infor. Manag.*, vol. 35, no. 2, Apr. 2015, pp. 137–144.

[4] M.S. Khan, F. Coenen, D. Reid, R. Patel, and L. Archer, "A Sliding Windows Based Dual Support Framework for Discovering Emerging Trends from Temporal Data," *Knowl.-Based Syst.*, vol. 23, no. 4, May 2010, pp. 316–322.

[5] G. Dong and J. Li, "Efficient Mining of Emerging Patterns: Discovering Trends and Differences," In *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery Data Mining*, San Diego, CA, USA, Aug. 15–18, 1999, pp. 43–52.

[6] G. Cormode and S. Muthukrishnan, "What's New: Finding Significant Differences in Network Data Streams," *IEEE/ACM Trans. Netw.*, vol. 13, no. 6, Dec. 2005, pp. 1219–1232.

[7] H. Alhammady and K. Ramamohanarao, "Mining Emerging Patterns and Classification in Data Streams," In *IEEE/WIC/ACM Int. Conf. Web Intell.*, Compiegne, France, Sept. 19–22, 2005, pp. 272–275.

[8] Y. Chen, G. Dong, J. Han, B.W. Wah, and J. Wang, "Multi-dimensional Regression Analysis of Time-Series Data Streams," In *Proc. Int. Conf. Very large Data Bases*, Hong Kong, China, Aug. 20–23, 2002, pp. 323–334.

[9] V.E. Lee, R. Jin, and G. Agrawal, "Frequent Pattern Mining in Data Streams," In *Frequent Pattern Mining*, New York, USA: Springer, 2014, pp. 199–224.

[10] C. Giannella, J. Han, J. Pei, X. Yan, and P.S. Yu, "Mining Frequent Patterns in Data Streams at Multiple Time Granularities," In *Data Mining: Next Generation Challenges and Future Direction*, MIT/AAI Press, 2004, pp. 191–212.

[11] T.M. Akhriza, Y.H. Ma, and J.H. Li, "A Novel Fibonacci Windows Model for Finding Emerging Patterns over Online Data Stream," In *Int. Conf. Cyber Security Smart Cities, Ind. Contr. Syst. Commun.*, Shanghai, China, Aug. 2015, pp. 1–8.

[12] P. Kralj, N. Lavrač, D. Gamberger, and A. Krstačić, *Contrast Set Mining for Distinguishing between Similar Diseases*, Lecture Notes in Computer Science, vol. 4594, New York, USA: Springer, 2007.

[13] J.Y. Li, H. Liu, S.K. Ng, and L. Wong, "Discovery of Significant Rules for Classifying Cancer Diagnosis Data," *Bioinformatics*, vol. 19, sup. 2, Sept. 2003, pp. 93–102.

[14] J.Y. Li, H. Liu, J.R. Downing, A.E. Yeoh, and L. Wong, "Simple Rules Underlying Gene Expression Profiles of More than Six Subtypes of Acute Lymphoblastic Leukemia (ALL) Patients," *Bioinformatics*, vol. 19, no. 1, Jan. 2003, pp. 71–78.

[15] L. J Chen and G.Z. Dong, "Masquerader Detection Using OCLEP: One Class Classification Using Length Statistics of Emerging Patterns," In *Int. Conf. Web-Age Inform. Manag.* Workshops, Hong Kong, China, June 17–19, 2006, p. 5.

[16] C. Borgelt, "Data Mining and Knowledge Discovery," *WIREs Mining Knowl. Discovery*, vol. 2, no. 6, Oct. 2012, pp. 437–456.

[17] X. Chen and Z. Liu, "Finding Contrast Patterns in Imbalanced Classification Based on Sliding Window," In *Proc. Int. Conf. MMME*, Beijing, China, Dec. 30–31, 2016, pp. 161–166.

[18] H.F. Li and H.S. Chen, "Discovering Emerging Melody Patterns from Customer Query Data Streams of Music Service," In *IEEE Int. Conf. Multimedia Expo*, Barcelona, Spain, July 11–15, 2011, pp. 1–4.

[19] J. Bailey and E. Loekito, "Efficient Incremental Mining of Contrast Patterns in Changing Data," *Inform. Process. Lett.*, vol. 110, no. 3, Jan. 2009, pp. 88–92.

[20] C. Lee, C. Lin, and M. Chen, "Sliding-Window Filtering: an Efficient Algorithm for Incremental Mining," In *Proc. Int. Conf. Inform. Knowl. Manag.*, Atlanta, GA, USA, Oct. 2001, pp. 263–270.

[21] H-f. Li and S-Y. Lee, "Mining Frequent Itemsets over Data Streams Using Efficient Window Sliding Techniques," *Expert Syst. Applicat.*, vol. 36, no. 2, Mar. 2009, pp. 1466–1477.

[22] J.H. Chang and W.S. Lee, "Finding Recent Frequent Itemsets Adaptively over Online Data Streams," In *Proc. ACM. SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Washington, DC, USA, Aug. 24–27, 2003, pp. 487–492.

[23] J.H. Chang and W.S. Lee, "estWin: Online Data Stream Mining of Recent Frequent Itemsets by Sliding Window Method," *J. Inform. Sci.*, vol. 31, no. 2, Apr. 2005, pp. 76–90.

[24] G.S. Manku and R. Motwani, "Approximate Frequency Counts over Data Streams," In *Proc. Int. Conf. Very Large Data Bases*, Hong Kong, China, Aug. 20–23, 2002, pp. 346–357.

[25] J. Cheng, Y. Ke, and W. Ng, "A Survey on Algorithms for Mining Frequent Itemsets over Data Streams," *Knowl. Inform. Syst.*, vol. 16, no. 1, July 2008, pp. 1–27.

[26] Machine Learning Repository, "Online Retail Data Set," Accessed May 29, 2017. http://archive.ics.uci.edu/ml/datasets/online+retail

Tubagus Mohammad Akhriza is currently with Pradnya Paramita School of Informatics Management and Computer, Malang, Indonesia, as the rector. He received his BS degree in mathematics from Brawijaya University, Indonesia (1996), and his MS degree in management information systems (2001) from Gunadarma University, Indonesia (2001). He completed his doctoral study in communication and information system engineering at Shanghai Jiao Tong University, China (2015), and received his PhD degree in information technology from Gunadarma University, Indonesia (2016). His interests are in data mining and business intelligence.

Yinghua Ma received her BSc degree in power system automation (1993) and her MS degree in electronics (1996) from Shangdong University, Jinan, China. She received her PhD degree in communication and information systems (2004) from Shanghai Jiao Tong University, China. In 2005, she was at Burgundy University, Dijon, France as a postdoctoral researcher working mainly on semantic networks. She is currently a lecturer in the School of Network Security Engineering at Shanghai JiaoTong University. Her research interests are in information semantic analysis.

Jianhua Li is a professor, doctoral supervisor, and executive sub decanal of the School of Information Security Engineering at Shanghai Jiao Tong University, is the director of the Shanghai Key Laboratory of Integrated Administration Technology for Information Security and Network Information Security Management and the Service Engineering Research Center of the State Ministry of Education. He received his BS, MS, and PhD degrees in electronic engineering from Shanghai Jiao Tong University, China in 1986, 1991, and 1998, respectively. Since 2000, he has been the chief expert and management expert of the Information Security Technology Panel for National Project 863 during the 10th Five-Year Plan Period, a member of the National E-government Pilot Demonstration Project Total Panel, an expert in the China E-government Standardization Total Group, a member of the National Information Security Standardization Expert Committee, a consultant for the Administration for the Protection of State Secrets, and the leader and chief expert of the Total Group in National Information Security Application Demonstration Project (S219) Stage II.