Data & Knowledge Engineering 89 (2014) 21–37

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Editorial

Mining frequent itemsets in data streams within a time horizon

Luigi Troiano, Giacomo Scibelli

University of Sannio, Dept. of Engineering, Viale Traiano, 82100 Benevento, Italy

ARTICLE INFO

Article history:
Received 18 August 2011
Received in revised form 22 October 2013
Accepted 22 October 2013
Available online 27 December 2013

Keywords:
Data mining
Mining methods and algorithms
Frequent itemsets

ABSTRACT

In this paper, we present an algorithm for mining frequent itemsets in a stream of transactions within a limited time horizon. In contrast to other approaches that are presented in the literature, the proposed algorithm makes use of a test window that can discard non-frequent itemsets from a set of candidates. The efficiency of this approach relies on the property that the higher the support threshold is, the smaller the test window is. In addition to considering a sharp horizon, we consider a smooth window. Indeed, in many applications that are of practical interest, not all of the time slots have the same relevance, e.g., more recent slots can be more interesting than older slots. Smoothness can be determined in both qualitative and quantitative terms. A comparison to other algorithms is conducted. The experimental results prove that the proposed solution is faster than other approaches but has a slightly higher cost in terms of memory.

1. Introduction

When we search for interesting patterns in sets, sequences and relations of data, the identification of target itemsets, i.e., subsets of occurring items, represents the starting point of most of the analyses. As mined itemsets can reveal interesting and previously unknown facts [1–8], the learning of recurrent patterns in data streams is challenging because of the dynamic nature of the problem and the additional real-time constraints.

Data streams occur in many examples of practical interest, such as in sensor data generated by sensor networks, online transactions recorded by the retail chains, grocery sales data, visited pages and click-streams in web applications, calls and instant messages that are placed in telecommunication networks, daily weather/traffic records, stock market data, or performance measurements in network traffic management and monitoring. The mining of flowing data offers new challenges, because data streams are continuous and unbound, they require to be processed considering real-time constraints and data distributions which change over time. In the data streams, some of the patterns become more frequent, while other patterns tend to disappear. As motivating examples we outline the following two cases: the visited web pages and the rainfall data. In the first case, we are interested to analyze the web traffic generated by single visits in order to identify which pages are mostly visited over time. In this case, items are web pages or site sections, and transactions collect which pages/sections are visited by a unique visitor. Records have different sizes. Frequent itemsets represent the groups of pages/sections that are mostly visited within a limited time horizon. In the second case, we aim to analyze how rainfalls are changing by time over an area of interest. Here, items are the rainfall levels recorded over the time at different locations within a geographic area. Records have the same size. Frequent itemsets relate to groups of rainfall levels that are mostly occurring within a time period.

Both cases will be used as domains of application as part of our experimentation.

The time horizon provides a frame of interest in which to search for the frequent itemsets. The contribution of this paper is in presenting the Window Itemset Shift (WIS) algorithm, which provides an efficient solution to the search for frequent itemsets in a

E-mail addresses: troiano@unisannio.it (L. Troiano), scibelli@unisannio.it (G. Scibelli).

0169-023X/$ – see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.datak.2013.10.002

stream of transactions when a time horizon is given, specifically, when this horizon is smoothly defined by fuzzy sets or determined according to stream characteristics.

The simple iterated structure of a standard algorithm is inefficient, because the search begins from scratch at each iteration and does not account for any memory of the transaction stream. This case is the Apriori [1] algorithm, when it is re-iterated as new transactions are entering the frame of interest. Other approaches have been proposed in the literature. Although the literature contains some optimizations to Apriori, their logic is based on re-iterating the algorithm at each step.

However, frequent itemsets must occur in a restricted portion of time. If not, the support does not reach the threshold. With respect to this property, an initial idea is to consider the minimal window that makes it possible to discard those itemsets that are certainly not frequent. In addition, because most of the itemsets are held within the time frame of interest, a further enhancement is to maintain a memory of itemset candidates that might become frequent and to update this list as new transactions are entering and old transactions are exiting the time frame. As a consequence, the number of candidates to consider at each step is smaller than those processed by other approaches. Finally, the support counting is limited to the test window and it does not require a pass through the dataset. This approach, as shown by experimentation on datasets with different characteristics, allows the algorithm to perform better than the other algorithms proposed in the literature.

Because the time slots might have a different relevance within the time horizon considered, we can assign a degree of interest to each of them. This approximation leads to "fuzzify" the algorithm in such a way that the support count is affected by the different relevances assumed by the transactions as they move along the time frame. Although this feature does not provide any further enhancement to the performances, it expands the meaningfulness of the algorithm, enabling a higher expressivity. In addition, as explained in Section 4, this feature allows us to address with the unbounded windows with a limited capacity and improves the algorithm's scalability.

This paper is organized as follows: Section 2 introduces the definitions and the properties of frequent itemsets for the non-familiar reader, Section 3 provides a brief overview of the related research, Section 4 introduces the concepts that underlie the WIS algorithm, Section 5 describes the algorithm's logic and architecture, Section 6 compares WIS to other algorithms by experimentation on datasets that have different characteristics, and Section 7 outlines the conclusions.

2. Preliminaries

A group or set of items entailed by database records, e.g., the set of items that a customer collects in a market basket, is referred to as an *itemset*. More formally, let $X = \{i_1,i_2,\ldots,i_m\}$ be a set of m distinct literals called *items*. Let the dataset $\mathcal{D}$ be a collection of transactions over X. Each transaction (or record) $r \in \mathcal{D}$ contains a subset of items, i.e., $r \equiv \{i_i,i_j,\ldots,i_k\} \subseteq X$. The number of items in r provides its *length*. The transactions in $\mathcal{D}$ can entail different lengths. A subset of items $I \subseteq X$ is called an *itemset*.

The number of times that an itemset I occurs in the transactions is the *support count* of I, which is denoted as $supp(I)$. Frequent itemsets are defined with respect to the support threshold S; as a result an itemset I is frequent if $supp(I) \geq S$.

The itemset support count is related to the support count of its subsets and supersets. In fact, given two itemsets I and J such that $I \subset J$, the number of times that I occurs is at least the number of occurrences of J because the former is part of the latter. Therefore,

$$supp(I) \geq supp(J), \quad \forall I \subset J \subseteq X \tag{1}$$

In addition, it is useful to further classify an itemset X as being *maximal frequent* if X is frequent, but any $Y \supset X$ is not [9].

The problem of counting the number of distinct itemsets in a dataset, given an arbitrary support threshold, is NP-complete, and the problem of mining itemsets is NP-hard [10]. If $|X|$ is the cardinality of X, then the number of possible distinct itemsets is $2^{|X|}$. To reduce the combinatorial search space, most of the algorithms exploit the following two properties:

- *Downward closure*: all of the subsets of a frequent itemset are frequent[1]
- *Anti-monotonocity*: supersets of an infrequent itemset must be infrequent, too.

3. Related research

3.1. Mining of frequent itemsets

Traditionally, itemset mining is associated with the discovery of association rules [2,11] because they provide a subset of patterns within which to search. The problem of mining association rules in large databases can be divided into two subproblems: (i) seeking frequent itemsets; and (ii) discovering association rules among the found itemsets.

The first and most noticeable algorithm for mining frequent itemsets is known as *Apriori*, which was proposed independently by Agrawal and Srikant [3] and Mannila, Toivonen and Verkamo [12] and was later joined in [4]. Apriori is a levelwise, breadth-first, bottom-up algorithm, as outlined by Algorithm 1.

[1] The name of the property comes from the fact that the set of frequent itemsets is closed with respect to set inclusion.

Algorithm 1 Apriori

L_k: frequent itemsets of size k
G_k: generated itemsets of size k
ml: size of the longest transaction
S: the frequent itemset minimal threshold

find L_1
$k = 1$
while $L_k \neq \emptyset$ and $k < ml$ **do**
  $G_{k+1} = L_k$ join L_k
  $L_{k+1} = \{I \in G_{k+1} | supp(I) \geq S\}$
  $k = k + 1$
end while
return $\bigcup_k L_k$

The main idea is to first perform a database scan to count the supports of *1-itemsets*, which are itemsets made of single items. After selecting frequent 1-itemsets, the supersets, called *candidates*, are generated and evaluated, while passing through the database again. This process is iterated until no new candidate is generated. Therefore, at step *k*, frequent itemsets that have *k* items are available in list L_k. Each step is composed of two phases. First, $(k + 1)$-itemsets are generated only from elements in L_k and are placed in G_{k+1}. The generation of candidates is performed by *joining* L_k with itself. This operation comprises merging two itemsets $I_1, I_2 \in L_k$, which share *k*-1 items in common. The resulting itemset $I = I_1 \cup I_2$, which is made of $k + 1$ items, is placed in G_{k+1}.

Next, their support is computed by scanning the dataset and discarding those itemsets whose support is below the minimal threshold *S*. This approach entails a number of passes through the dataset, where the number of passes equals the length *ml* of the largest transaction. The result of each step is the list of frequent itemsets L_{k+1}, which will be used in the following steps. The algorithm stops when L_{k+1} is empty or when $k = ml$. The algorithm output is obtained by merging lists of frequent itemsets L_k with $k = 1 \ldots ml$. The traversal by levels performed by Apriori is based on downward closure and anti-monotonocity properties.

The main shortcoming of Apriori is the time that is consumed when passing through the dataset at each of the iterations; thus, several enhancements have been proposed to overcome this limitation. Among them, we note *Partition*, which was proposed by Savasere, Omiecinski and Navathe [13]; Partition minimizes the I/O by scanning the dataset only twice. This algorithm performs a dataset partition, dividing the dataset into smaller chunks that are handled sequentially in memory. In the first pass, it generates the set of all potentially frequent itemsets. In the second pass, the overall support of each itemset is computed. Other contributions in this area are found in [14–22].

3.2. Frequent itemset mining in data streams

The main challenge in mining frequent itemsets in data streams is to enumerate them at a rate that is compatible with the speed at which the transactions are presented. This goal requires algorithms with in-memory data structures and a minimal (possibly one) pass through the dataset. To face these constraints, specific attention has been paid to the efficient mining of frequent itemsets in data streams. According to [23], we can group the approaches into three main categories: (i) landmark-, (ii) *damped*- and (iii) *sliding*-window based mining.

In landmark algorithms, itemsets are counted on the transactions between a specific timestamp, the landmark, and the present. Thus, in landmark algorithms, records are ongoing in the frame of interest.

Li, Lee and Shan investigated the problem of mining all frequent itemsets [24] and maximal frequent itemsets [25] in a landmark window. Both *DSM-FI* and *DSM-MFI* variants of their algorithm are based on a singlepass support count of streaming data and on a compact prefix tree-based pattern representation called *summary frequent itemset forest* (SFI-forest). The algorithm provides an incremental update of the SFI-forest by projecting each forthcoming transaction into a set of sub-transactions and pruning those elements that result in infrequent itemsets. Thus, all frequent itemsets that are embedded by the transaction stream can be determined by the current SFI-forest. They argue that the proposed approach provides a stable memory footprint and a time-efficient solution based on one-pass support counting.

The latter aspect is also considered by Ao et al. [26]. They propose *FpMFI-DS*, a one-pass algorithm based on *FpMFI* [27]. In contrast with FpMFI, their solution performs a single pass over the data stream to build the FP-tree. When the transactions enter

the window, embedded items are inserted into the FP-tree following a lexicographical order, whereas when they come out, they are deleted. The pruning of infrequent sub-trees is optimized.

The mining of frequent itemsets is usually faced by looking for an exact answer to the question of what are the most often occurring patterns in the data stream. Approximation can provide a rough but faster solution, which can face real-time constraints in data streaming.

Manku and Motwani [28] developed two single-pass algorithms, namely *Sticky Sampling* and *Lossy Counting*, for mining items (and itemsets) in a landmark window. Sticky Sampling provides an approximation of the resulting elements, linking the probability of being considered to the occurrence in the data stream. Lossy Counting provides an approximation of support, by grouping elements into buckets of a given size and support. To apply Lossy Counting to the mining of frequent itemsets, the authors proposed TRIE, a lattice-based in-memory data structure, which is used to store itemsets and approximate supports.

In contrast to landmark models in which all of the itemsets are treated in the same way in spite of how recent they are, in damped window models, recent itemsets are more relevant than older ones, i.e., the latter provides a lower contribution to the itemset support computation.

As an example, Chang [29] proposed *estDec*, which is an algorithm that assigns a weight to each transaction that decreases by age. Itemsets are kept by a prefix-tree lattice structure called a *monitoring lattice*. Two operations, *delayed insertion* and *pruning*, are used to efficiently update the data structure and to keep the footprint light. Specifically, the delayed insertion is to find any new itemset that has a high possibility of becoming frequent in the near future.

In a sliding window model, knowledge discovery is performed over a fixed number of recently generated transactions, in such a way that the frame of interest looks as though it is moving along the data stream. If the frame accounts for a fixed number of transactions, then the sliding window is said to be *transaction-sensitive*. Otherwise, if the frame is related to a fixed time interval, the window is said to be *time-sensitive*. In the case of regular data streams, in which new transactions flow in at a given frequency, the two sub-problems coincide.

Lee, Lin and Chen [30] proposed a sliding-window filtering (SWF) algorithm. The sliding window is made of a sequence of partitions. Each partition entails a number of transactions. The window is moved forward by partitions in such a way that the oldest partition is disregarded and a new partition enters the frame of interest. SWF uses candidate 2-itemsets as a basis for generating the whole set of candidates. Therefore, a 2-itemset candidate list is associated with each partition. A filtering threshold is applied to each partition to rule the candidate itemsets' generation. SWF requires scanning the whole sliding window to mine the frequent itemsets, which is the main drawback of this approach.

The problem of considering efficient in-memory data structures have been considered by other authors. Chang and Lee [31] proposed the *SWFI-stream* for finding frequent itemsets within a transaction-sensitive sliding window. Similar to the approach proposed by Manku and Motwani [28], SWFI-stream relies on a prefix tree lattice structure called the *monitoring lattice*. This approach is accompanied by another in-memory data structure called the *current transaction list* (CTL), which is used to maintain all of the transactions within the range of the current sliding window. Both of the structures are updated as the window moves forward.

Following a similar approach, Chi et al. [32] proposed *Moment*, an algorithm that is aimed at mining closed frequent itemsets within a transaction-sensitive sliding window. These authors adopt a prefix-tree-based data structure, which is called the *closed enumeration tree* (CET), to refer to a dynamic set of itemsets. More specifically, four types of nodes are considered: infrequent gateway nodes, unpromising gateway nodes, intermediate nodes and closed nodes. This information is used by Moment to traverse the CET structure when a new transaction comes into the frame of interest but also when the oldest transaction leaves the frame. Therefore, traversing the CET can be time consuming.

More recently, Li and Lee [23] proposed two one-pass algorithms, *MFI-TransSW* and *MFI-TimeSW*; the first algorithm is transaction-sensitive, and the second algorithm is time-sensitive. Both variants rely on an effective bit-sequence representation of items, assuming the value of 1 in correspondence to transactions in which the item is included. They perform three phases. The first phase, called *window initialization*, is composed of transforming each item of the new incoming transaction into its bit-sequence representation. The second phase, called *window sliding*, is composed of removing the oldest transaction from the window. The third phase, called *frequent itemset generation*, is composed of generating the set of candidate itemsets from the frequent itemsets of the previous step, in a fashion similar to Apriori.

4. Frequent itemsets within a limited window

Intuitively, a window is a time frame that is of interest for the analysis. In practice, the frame of interest is a time interval. The window *limitedness* depends on its lower bound, which, if not limited, leads the analysis to consider an increasing number of elements because of new records that are added; however, the oldest are not discarded.

Definition 1. Limited window

A limited window has a finite lower bound. □

When a stream of transactions occurs, records flow across a limited time window, as depicted in Fig. 1. As a result, the support of itemsets is going to change according to which records are coming into the window and which records are leaving the window.

Such a flow can occur record by record or by groups of records. This approach depends on the number of transactions that flow in and out at each iteration. Moving one slot forward, a group of transactions enters the scope of the window while another

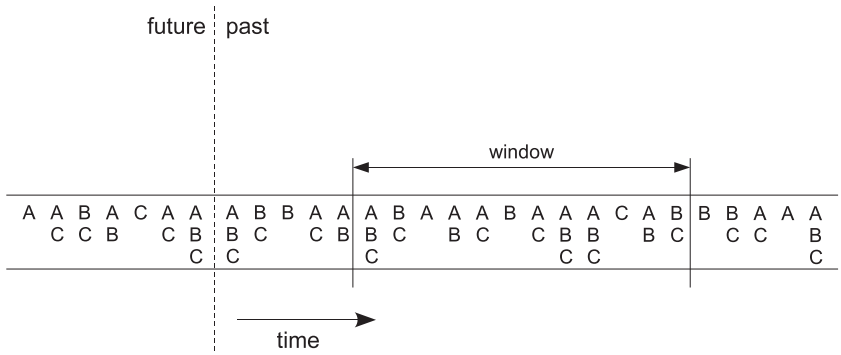

Fig. 1. A stream of transactions flowing across a time window.

leaves. For the sake of simplicity, we will assume that the number of elements considered within the window does not change over time. Thus, the number of records that flow into the window is equal to the number of records that flow out.

The number of elements considered within a window represents the *memory capacity* that is offered by the window. Limited windows have a limited capacity. Unlimited windows might entail a capacity that goes on ad infinitum. This case involves windows whereas each element is considered to have the same relevance as the others.

However, in practice, elements can be accounted for by a different degree of interest, as is accomplished by some of the algorithms illustrated earlier. This approach leads the window to fade at its edges, entailing a smooth window[2] such as that depicted in Fig. 2.

The sum of the degree of interest $\mu(t_i)$ by which each time-slot t_i is assumed within the window W, a.k.a., (σ-count) in fuzzy logic, provides the window memory capacity C, which is defined as

$$C(W) = \sum_{t_i \in W} \mu(t_i). \tag{2}$$

Obviously, windows can be *continuous* or *discrete* in time, and *causal* (if all of the elements refer back to past events) or *anti-causal*. For the intent of this paper, we will consider only time-discrete causal windows.

Convexity is an important characteristic of smooth windows. A smooth window is convex if the function that describes the element's degree of interest is such that, at each level of interest, the time slots that entail an interest over that level constitute an interval. Formally, we provide the following definition:

Definition 2. Convexity

A smooth window W is convex iff

$$min\left(\mu(t_i), \mu\left(t_j\right)\right) \leq \mu(t_k) \leq max\left(\mu(t_i), \mu\left(t_j\right)\right) \forall t_i < t_k < t_j \in W. \tag{3}$$

□

Smooth windows can be unbound but still have a limited capacity. This circumstance is possible when assuming that the degree of interest decreases superlinearly, as depicted in Fig. 3. In this case, there is an exponentially decreasing function. Formally, we have the following definition:

Definition 3. Asymptotically limited window

A smooth window is said to be asymptotically limited if its computed capacity has a limit that exists and is finite. In other words, iff

$$C(W) = \lim_{t_i \to \infty} \sum_{t_i \in W} \mu(t_i) = c \in]0, +\infty[\tag{4}$$

□

In this paper, we are interested in limited windows. The memory capacity of a given window offers an upper bound to the minimum support threshold S that is used to select the frequent itemsets. Indeed, we need $S \leq C(W)$ to obtain a non-empty resulting set. Under this constraint, we can find itemsets that are over this threshold and others that are below. The following necessary condition helps to test if an itemset I is possibly frequent or not.

[2] Some readers could object that this construct is formally equivalent to a fuzzy window. Although this equivalence is true from a formal point of view, we note that the window smoothness can be determined in different ways. Besides a subjective approach by which we assign arbitrarily a different degree of relevance to each time slot, we will show that it is possible to build a smooth window according to the quantitative characteristics of the stream.

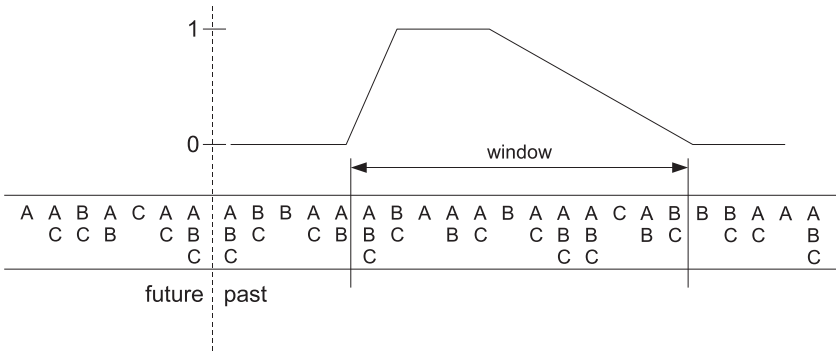

Fig. 2. A smooth (limited) window.

Proposition 1. Given a window W and a minimum support threshold S, such that $S \leq C(W)$. A frequent itemset I, i.e., $supp(I) \geq S$, must necessarily appear in a transaction taken among a subset of points $W_S \subseteq W$ such that

$$C(W_S) = \sum_{t_i \in W_S} \mu(t_i) > C(W) - S \tag{5}$$

Proof. The proof is straightforward. Indeed, if I does not appear in W_S, then we have

$$supp(I) = \sum_{t_i \in W | I \in t_i} \mu(t_i) = \sum_{t_i \in W \; W_S | I \in t_i} \mu(t_i) \leq$$
$$\leq \sum_{t_i \in W \; W_S} \mu(t_i) = C(W) - C(W_S) < S$$

which contradicts the hypothesis that I is frequent. □

The condition given by Proposition 1 provides a test for discarding those itemsets whose support is below the threshold; thus, it filters out candidates that cannot be frequent. In addition, it helps to address unbound windows that have a limited capacity $C(W)$, although the itemset support count should be approximated by the limit.

Example 1. Let us consider the case that is depicted in Fig. 4. The memory capacity of the window is $C(W) = 12$. If we assume that there is a support threshold $S = 8$, then we need a test window such that $C(W_S) > C(W) - S = 4$. Because the interest degree in W is equal to 1 for any of the elements, the smallest test window is composed of 5 elements, of any choice. For the sake of simplicity, we choose a window that is composed of consecutive elements within the interval [2–6]. This window contains transactions {*A*,*AB*,*AC*,*B*, *AC*}. Frequent itemsets are required to appear in W_S. The set of candidate itemsets is $C = \{A,B,C,AB,AC\}$; as a result, itemsets {*BC*,*ABC*} can be discarded because they are certainly not frequent. Indeed, their support is respectively $supp(BC) = 5$ and $supp(ABC) = 3$.

The efficiency of the test depends on the number of transactions to inspect in order to find a given itemset. This number becomes smaller by increasing the minimum support threshold S, because fewer transactions are required to overcome the quantity $C(W) - S$. However, given the value of S the following criterion assures a minimal number of transactions.

Proposition 2. Given the windows W and a support threshold S, the window subset W_S, that is composed of the largest m elements w.r.t. the degree of interest μ has the minimal number of elements that can overcome the quantity $C(W) - S$.

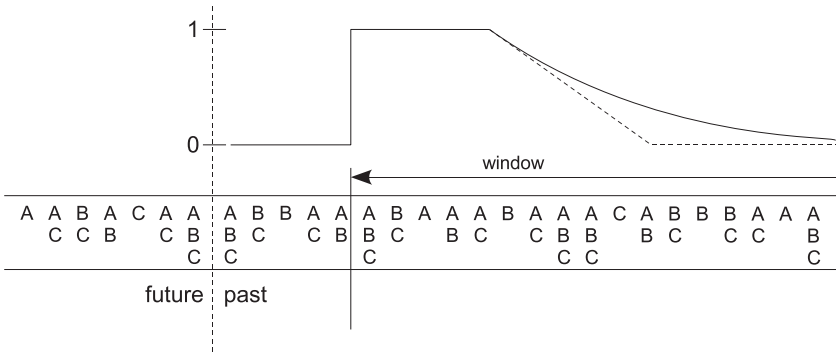

Fig. 3. An asymptotically limited window.

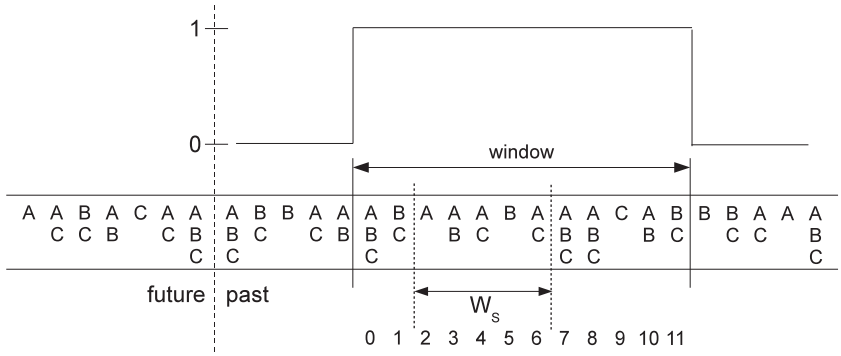

Fig. 4. Example with a sharp window.

Proof. Let $(\cdot)$ be a decreasing ordered permutation of the elements $t_1,\ldots,t_n \in W$ w.r.t. the degree of interest μ, such that $\mu(t_{(1)}) \geq \ldots \geq \mu(t_{(n)})$. Let $W_S \equiv \{t_{(1)},\ldots,t_{(m)}\}$ be able to overcome the quantity $C(W) - S$. Then, $C(W_S) > C(W) - S \geq C(W_S \backslash \{t_{(m)}\})$. Let us assume that W_S is not the subset that has the minimal number of transactions that can cover the quantity $C(W)-S$. Thus, there is at least another W_S that is composed of fewer elements, with no more than *m-1* elements. However, because $W_S \backslash \{t_{(m)}\}$ is made of the $m - 1$ elements that have the largest degree, we get

$$C\left(W_S^{'}\right) = \sum_{t_i \in W_S^{'}} \mu(t_i) \leq \sum_{t_i \in W_S \backslash \{t_{(m)}\}} \mu(t_i) \leq C(W) - S$$

Thus, we are not able to overcome the quantity $C(W) - S$ as required. □

If W is convex, then the optimality criterion given by Proposition 2 suggests taking an element with the maximum degree of interest and moving to the left and right side of it, until the quantity $C(W) - S$ is reached. This approach leads to obtaining an optimal convex subset W_S for testing itemsets, as depicted in Fig. 5. A different choice of slots would lead to considering larger test windows, because the larger number of transactions should be accounted for to cover the quantity $C(W) - S$, which potentially enlarges the number of candidates to consider. This approach is described by the following example.

Example 2. Let us consider the case that is depicted in Fig. 2. We have $C(W) = 12$; as a result, if we assume that $S = 8$, then we can select any W_S such that $C(W_S) > 4$. A first choice could be to use the window tail $W_S \equiv [3–11]$, as outlined in Fig. 6. Transactions falling into W_S are {*AB*,*AC*,*B*,*AC*,*ABC*,*ABC*,*C*,*AB*,*BC*} in such a way that the candidate list is {*A*,*B*,*C*,*AB*,*AC*,*BC*,*ABC*}, which represents the whole itemset lattice. This choice is not optimal. We would expect that choosing a smaller window will lead us to a smaller set of candidates. Indeed, if we choose $W_S \equiv [1–6]$ according to the method outlined in Proposition 2, then the candidate list is reduced to {*A*,*B*,*C*,*AC*,*BC*}, as depicted in Fig. 7. Specifically, the itemsets {*BC*,*ABC*} can be discarded as certainly being not frequent.

The variant *Apriori with Window Test* (AWT), which is outlined by Algorithm 2, accounts for the window test, as suggested by Propositions 1 and 2. In this case, the mining of frequent itemsets is also driven by both conditions (i) $\exists t_i \in W_S : I \subseteq t_i$ and (ii) $supp(I) \geq S$ (see line 10). Because the support condition (ii) entails the occurrence condition (i), it could appear to be redundant to consider both. However, testing the occurrence condition (i) first is a faster way to filter out non-frequent itemsets, which proves to be a shortcut to the conjunction that is expressed in line 10. This characteristic can be exploited to build a faster online mining algorithm, as shown in the following section.

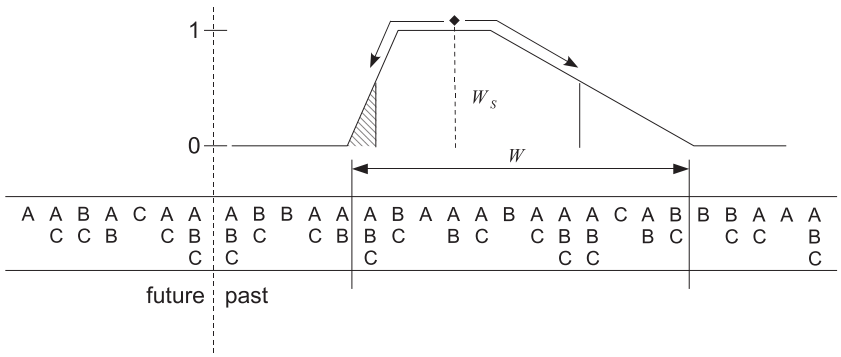

Fig. 5. Optimal convex test window.

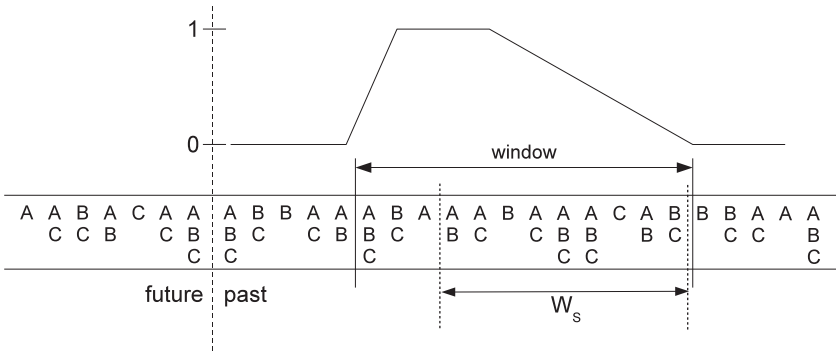

Fig. 6. Example with a smooth window and a random test window.

A smooth window offers additional possibilities compared with a sharp window. On the qualitative side, a smooth window can be determined subjectively by assigning a different relevance degree to each slot. In this way, some slots (usually the most recent) are assumed to be more relevant than the others (usually the oldest), which are less interesting to us. This approximation leads to shaping a fuzzy set over the window, where the membership degree reflects the importance of the slot when counting occurrences. We call this property *interpretability*.

Besides this property, on the quantitative side, a smooth window allows us to reduce the width of the optimal test window, given the same time horizon. Indeed, we recall that the test window W_S must account for as many slots as required to cover the capacity $C(W_S) = C(W) - S$. If W is smooth then $C(W)$ is smaller than when W is sharp. Thus, given the minimum support threshold S, $C(W_S)$ is smaller. We call this property *fitness*.

Algorithm 2 Apriori with Window Test (AWT)

L_k: frequent itemsets of size k
G_k: generated itemsets of size k
ml: size of the longest transaction

find W_S
find L_1
$k = 1$
while $L_k \neq \emptyset$ and $k < ml$ **do**
  $G_{k+1} = L_k$ join L_k
  $L_{k+1} = \{I \in G_{k+1} \mid \exists t_i \in W_S : I \subseteq t_i \wedge supp(I) \geq S\}$
  $k = k + 1$
end while
return $\bigcup_k L_k$

It appears clear that interpretability and fitness make it possible to consider an unlimited time horizon, that is impossible to manage in the case of a sharp window, because $C(W) = \infty$ in that case. The window fitness can also model the stream dynamics. For example, an itemset will flow in at different times. Usually, this event does not occur on regular basis. Instead, itemsets will appear at different inter-arrival times. Let us suppose that 90% of the times the inter-arrival time is within the interval [0,5) and 10% of the time is within [5,50). To capture S occurrence events, we should consider a time horizon length of $50 \cdot S$, which leads to a testing window W length of $50 \cdot S - S$ slots. However, 90% of the time it would suffice to consider a horizon $5 \cdot S$, and a testing window of $5 \cdot S - S$ slots, which is much smaller. A smooth window enables us to still consider a time horizon length of $50 \cdot S$, but a testing window that is much smaller.

Several methods can be employed to build a smooth window that fits characteristics of the data stream. This problem would require a deeper investigation and would be out of the scope of this paper. We will only outline the problem in general terms, leaving further research to provide more depth on this issue.

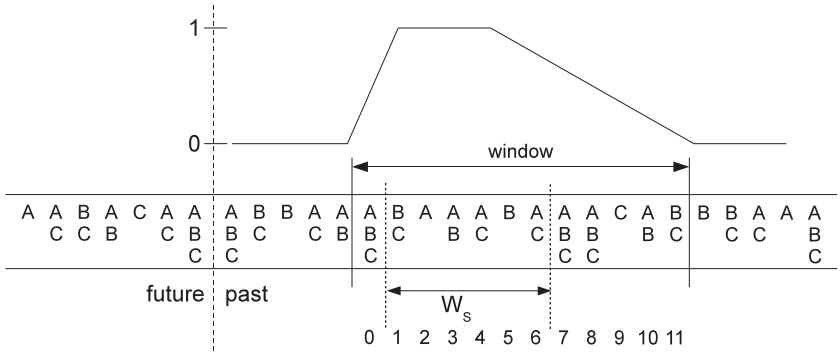

Fig. 7. Example with a smooth function and an optimal test window.

The most immediate approach is to assign a degree of relevance to each slot on a subjective basis. For example, it would be possible to consider what are the minimal and maximal horizons that we are interested in and to move from the first to the second linearly. This approach will lead to shape a trapezoidal window as shown in some of the examples above.

Other approaches could be related to the structural characteristics of the stream. A possible method can be outlined as follows:

1. Choose a collection of itemsets to which we are interested;
2. Observe the stream for a period of time in such a way that we are able to characterize it statistically;
3. Define the window width that can assure events with a given probability;
4. Compute the probability p_l of each itemset of interest to be frequent at the different possible lengths l of the window;
5. Assign at each length l a degree of relevance decreasing by p_l.

With respect to steps 3 and 4, a suitable distribution to use is Poisson. Indeed, a Poisson process is a simple stochastic process that counts the number of events, considering the time that these events occur and the given time interval of observation. A Poisson process is characterized by a parameter λ that corresponds to the average arrival rate of events. Arrivals are assumed to be independent (memorylessness), and governed by Poisson distribution

$$P[(N(b)-N(a)) = k] = \frac{e^{-\lambda_{a,b}}\left(\lambda_{a,b}\right)^k}{k!} \quad k = 0, 1, 2, \ldots \tag{6}$$

where $N(t+\tau) - N(t) = k$ is the number of events in the time interval $(t,t+\tau]$. In our case, events are itemsets of interest and k denotes their number of occurrences.

With respect to step 5, the degree of relevance decreases by a probability because the remaining part becomes less and less relevant to make the itemset frequent. This process should be performed for each itemset of interest. The overall window is shaped by choosing the maximum degree for each slot, in order to account for the characterization that is provided by each itemset.

The above description is only one example of a possible method to shape the window. Other methods are possible and are based on different criteria. For example, they could attempt to capture when the itemsets tend to appear and when they disappear, and to shape the window accordingly.

5. Online frequent itemsets update

Proposition 1 makes it possible to build an efficient strategy on which to perform online updates of frequent itemsets. As new records flow in, some itemsets become more frequent while others disappear.

Indeed, when new records flow into W, other records leave the window. Incoming records contribute to increasing the support of the itemsets, and leaving records contribute to reducing the support. At each step, we need a list C of candidate itemsets that can possibly become frequent as new records flow into W. Subsequently, the candidate list and support are updated at each step, according to which records enter or leave W. This approach is accomplished by the Window Itemset Shift (WIS) algorithm, which is outlined by Algorithm 3. According to the taxonomy provided in Section 3, WIS is a transaction-sensitive sliding-window based mining algorithm.

At the beginning of the algorithm, the optimal W_S is found (line 7), and the lists C and F are initialized by considering the records that are initially in W (lines 9–10). List C could be initially obtained by Algorithm 2 by considering only the occurrence condition, i.e., $C_{k+1} = \{I \in G_{k+1} | \exists t_i \in W_S : I \subseteq t_i\}$, thus relaxing the condition that is expressed at line 10. Indeed, the occurrence condition is necessary (but is not sufficient) to outline the frequent itemsets. However, although this approach looks appealing in theory, it is not feasible in many circumstances of practical interest. The only occurrence condition makes C explode in the number of candidates, which reaches, in some cases, the whole set of itemsets which is composed of 2^n elements, where n is the number of different items. A better option is to consider frequent itemsets as the initial candidates for the steps afterward. The online learning process that is

employed by WIS will include candidates that are not considered at the beginning. Algorithms that are devoted to mine frequent itemsets efficiently, such as FP-Growth [17] (applied to transactions within the sliding window), can improve the algorithm startup. However, this issue is outside of the focus of this work, and will not be considered further here.

For each itemset I, WIS makes use of an occurrence vector v_I, which counts the itemset occurrences for each sliding block in W, i.e., $v_I(r_i) = v$ if I is part of v records in r_i. In the simplest case of assuming one record for each time slot, the vector becomes binary (i.e., Boolean). To identify those itemsets that are quitting the testing region W_S, for each itemset $I \in C$, we maintain a memory of $last(I)$ which is the last position recorded for I within W_S. When a new record that contains I enters W_S, this value is reset to the first position; the result is that, when $last(I)$ reaches the W_S upper bound, I is discarded.

Algorithm 3 Window Itemset Shift (WIS)

C: candidate list
F: frequent itemsets
S: support threshold
W: frame of interest
S: support threshold

find optimal W_S

$F \leftarrow frequents(W, S)$
$C \leftarrow candidates(W, W_S, S)$
for all $I \in C$ **do**
  build vector v_I, $last(I) \leftarrow$ leftmost occurrence in W_S
end for

loop
  move W forward of h slots
  for all $I \in C$ **do**
    $last(I) += h$
    if $last(I) > length(W_S)$ **then**
      remove I from C
    end if
  end for
  $R_S \leftarrow$ records entering W_S
  for all $I_r,\ I_r \neq \emptyset,\ I_r \subseteq r,\ r \in R$ **do**
    if $I_r \in C$ **then**
      $last(I_r) = position(r)$
    else
      insert I_r in C
      build vector v_{I_r}, $last(I_r) \leftarrow$ leftmost occurrence in W_S
    end if
  end for
  $F = \{I \in C \mid supp(I) \geq S\}$
end loop

An example of how WIS works is provided in Fig. 8. For the sake of simplicity, we consider only a regular flow of 1 record entering W and 1 record leaving W at each iteration. When a cluster of incoming records R flows into W, another group R_S of records flows into W_S, which makes new candidates appear while those leaving W_S disappear. For example, in Fig. 8, BC enters the

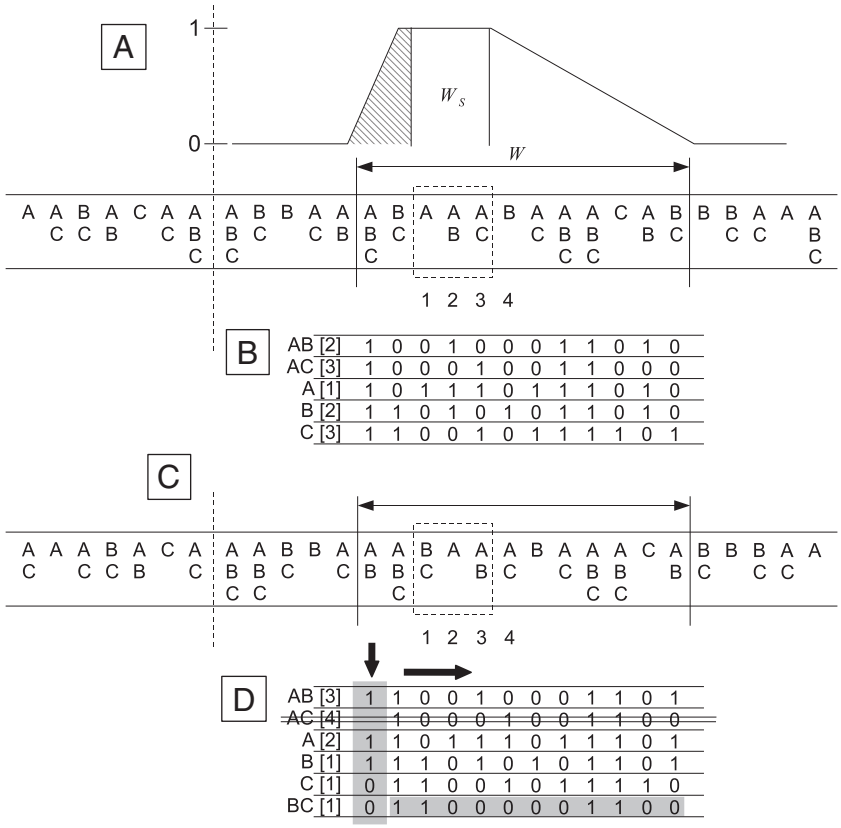

Fig. 8. WIS explained.

test window W_S while *AC* leaves W_S. This event produces a *right-shift* of the occurrence vector v_I for each itemset $I \in C$. When a record enters W_S, each non-empty subset $I_r \in 2^{R_S}$ becomes (if not yet) a candidate. This statement is true for each record $r \in R$. For example, *BC* becomes a candidate in Fig. 8. For those itemsets that are not yet included in *C*, the occurrence vector v_{I_r} is determined by looking at the current records in *W*. For the other itemsets, the number of times *I* occurs in *R* is stored in v_I. The support can be easily computed from v_I as:

$$supp(I) = \sum_{t_i \in W} v_I(t_i)\mu(t_i). \tag{7}$$

Those itemsets $I \in C$ such that $supp(I) \geq S$ are frequent and are recorded in the corresponding *F*.

6. Experimental results

To assess WIS performances and behavior, we performed a set of experiments that were aimed at testing WIS, under different operational conditions, versus conventional algorithms, i.e. Apriori and its variant AWT, when applied iteratively to a stream of transactions and other transaction-sensitive sliding window solutions, in particular MFI [23] and SWF [30].

The algorithms were all implemented in Java 1.5, and experiments were conducted on an Intel Core Duo 2.53 machine with 4 GB of RAM running on a Mac OS X 10.6.4 Snow Leopard.

To obtain an efficient execution, the itemsets were represented as a set of bits, which allowed for an efficient implementation of the bitwise operations, specifically the *and*, *or*, *shift* and *comparisons*, and an efficient memory occupation. Candidate and frequent lists are actually sets because they do not admit duplications. For the sake of simplicity, we assumed convex windows, in particular trapezoidal, triangular and sharp.

For each algorithm, it is possible to distinguish two phases: a *window initialization phase*, in which algorithms mine the frequent itemsets in the initial window, and a *window sliding phase*, in which the algorithms mine frequent itemsets while considering the new incoming transactions. The execution times reported in the tables below are referred only to the sliding window phase because the initialization window phase does not have great interest in a stream of data.

Table 1
Comparing WIS to Apriori and AWT.

			Support threshold 50%		Support threshold 10%	
Algorithm	Step	Iterations	Time (ms) (Ratio)	Memory (KB)	Time (ms)	Memory (KB)
T20I6D100K						
Apriori			16,183 (344×)	136,265.000	15,558 (17.0×)	136,435.960
AWT	1	20	13,238 (281×)	136,285.378	11,632 (12.7×)	131,449.832
WIS			47	130,790.272	913	128,794.264
Apriori			13,337 (16.3×)	137,040.200	15,097 (12.4×)	136,367.146
AWT	10%	20	11,305 (13.7×)	136,261.760	14,316 (11.7×)	136,466.058
WIS			820	131,604.392	1222	125,856.226
C20D10K						
Apriori			14,242 (2.24×)	29,251.712	>30 min.	
AWT	1	20	11,868 (1.86×)	29,171.426	>30 min.	
WIS			6370	25,900.560	372,087	420,457.720
Apriori			3137 (5.19×)	23,984.660	>30 min.	
AWT	10%	20	2587 (4.28×)	23,903.904	>30 min.	
WIS			604	25,799.744	521,713	783,490.051
R10KC20						
Apriori			229 (4.32×)	23,755.144	20,903 (40.4×)	24,493.906
AWT	1	20	207 (3.91×)	21,368.946	19,276 (37.3×)	26,223.944
WIS			53	22,415.744	517	21,736.992
Apriori			316 (3.09×)	21,393.786	20,699 (28.9×)	26,070.298
AWT	10%	20	219 (2.15×)	22,356.018	19,533 (27.3x)	24,908.600
WIS			102	23,806.578	716	21,820.152
R10KC15						
Apriori			2178 (16.8×)	22,357.792	>30 min.	
AWT	1	20	2077 (16.0×)	22,254.818	>30 min.	
WIS			130	23,136.778	16,996	106,595.186
Apriori			2419 (10.4×)	21,864.106	>30 min.	
AWT	10%	20	2504 (10.8×)	22,682.112	>30 min.	
WIS			232	21,518.704	22,654	189,042.464

6.1. Comparison to the Apriori approach

The purpose of this first part of experiments is to compare WIS to Apriori when the latter is merely reapplied at each iteration or when AWT filtering is applied. For test cases, we used datasets that had different characteristics of density and diversity. Specifically, for datasets that had a large number of items, we chose *T20I6D100K* [3] as an example of a sparse dataset and *C20D10K* [4] as an example of a dense dataset. The first choice is made of records that contain 893 different items, which are constructed according to the properties of market basket data that are typically weakly correlated, while the second choice is a census dataset from the PUMS (Public Use Microdata Sample) sample, which has 192 different items. In addition, we randomly generated a collection of sparse and dense datasets that were made of 20 different items. The first, *R10KC20*, has 10,000 records, the longest record having 20 different items; the second, *R10KC15*, is made of 10,000 records, each of which collects 15 different items. For each dataset, we performed two tests: the first test had a high support threshold (50%), and the second test had a low support threshold (10%). We observe that exceeding the threshold of 50% would not make sense for smooth windows, because this value is the upper limit for a triangular window capacity, i.e., $C(W) \leq 50\%$ of the records considered by W, and the trapezoidal windows hardly cover such a capacity. In any case, the window test contained 5% of all of the transactions, and the new incoming slide (i.e., iterations) contained 10% of the window transactions, to make the stream more consistent. We also considered two possible step values: when the stream moves 1 slot per time, and when it moves ahead the 10% of the window size. Experimental results are reported in Table 1.

WIS resulted in providing the lowest execution time at a cost of higher memory occupation. This relationship results from the data structures used, because WIS keeps an occurrence vector v_I for each candidate itemset.

An interesting and not fully expected result is the increasing gap between the Apriori and WIS execution time by reducing the support threshold. Indeed, by reducing the support threshold, we enlarge the size of the test window W_S. However, the cost for recomputing an exponentially larger set of frequent itemsets exceeds the cost that is paid by WIS in shifting v_I vectors and adding new candidate itemsets that will be at most 2^{ns}, where n_S represents the number of different items in the minimal window W_S,

[3] http://www.almaden.ibm.com/software/quest/Resources.
[4] http://www.census.gov/.

Table 2
Comparing WIS to MFI and SWF.

			Support threshold 50%		Support threshold 10%	
Algorithm	Step	Iterations	Time (ms) (Ratio)	Memory (KB)	Time (ms)	Memory (KB)
T20I6D100K						
MFI			11,784 (251×)	118,494.068	10,992 (12.0×)	109,388.157
SWF	1	20	11,116 (237×)	148,957.840	10,340 (11.3×)	148,563.640
WIS			47	130,790.272	913	128,794.264
MFI			10,002 (12.1×)	120,393.855	13,910 (11.4×)	119,280.373
SWF	10%	20	9944 (12.1×)	132,853.386	11,635 (9.52×)	148,865.712
WIS			820	131,604.392	1222	125,856.226
C20D10K						
MFI			9920 (1.56×)	21,938.374	>30 min.	
SWF	1	20	9259 (1.45×)	36,731.386	>30 min.	
WIS			6370	25,900.560	372,087	420,457.720
MFI			1025 (1.70×)	21,983.774	>30 min.	
SWF	10%	20	806 (1.33×)	27,850.872	>30 min.	
WIS			604	25,799.744	521,713	783,490.051
R10KC20						
MFI			198 (3.74×)	17,625.358	8,661 (16.8x)	19,839.958
SWF	1	20	571 (10.1×)	23,498.866	6.359 (12.3x)	30,431.512
WIS			53	22,415.744	517	21,736.992
MFI			177 (1.74×)	17,384.092	8.862 (12.4x)	21,759.855
SWF	10%	20	104 (1.02×)	15,281.312	6.192 (8.65x)	29,171.752
WIS			102	23,806.578	716	21,820.152
R10KC15						
MFI			1394 (10.7×)	20,009.482	>30 min.	
SWF	1	20	849 (6.53×)	22,606.778	>30 min.	
WIS			130	23,136.778	16,996	106,595.186
MFI			1233 (5.31×)	20,003.994	>30 min.	
SWF	10%	20	348 (1.50×)	22,630.858	>30 min.	
WIS			232	21,518.704	22,654	189,042.464

which is usually nS $\ll |I|$ In addition, the generation of candidates performed by Apriori at each level l is more expensive because it processes $\frac{nf_{l-1} * (nf_{l-1}-1)}{2}$, where nf_{l-1} is the number of frequent itemsets found at level $l-1$ (Table 2).

As expected, AWT slightly improves Apriori because the former still maintains the logic of the latter although it is applied to itemsets that occur in the smaller window test W_S, instead of in the whole window W. This is further confirmed by datasets used for experiments reported in Tables 3 and 5.

6.2. Comparison to the sliding-window algorithms

Among the different options that are presented in Section 3, we adopted an implementation of MFI and SWF, according to the pseudo-code provided, respectively, in [23] and [30]. Specifically, we considered MFI-TransSW because this code is the transaction-sensitive variant of MFI.

In this case, we considered only WIS with a sharp horizon, to obtain the same output of MFI and SWF.

Because SWF is based on dataset partitioning, this test case has been conducted using different sizes of the incoming slide. Specifically, we simulated two streams: the first stream has a single incoming transaction while the second stream has an incoming slide of 10% with respect to the initial window.

6.3. Comparison with real datasets

We tested WIS versus Apriori, AWT, MFI and SWF on the two real datasets mentioned as motivating examples in the introduction. The first stores the rainfall data that were collected monthly from January 1895 until December 2011.[5] We considered data on a 9-point grid that is centered near the Boston metropolitan area (Lat.:42.50 ± 0.50; Lon.:−71.50 ± 0.50). The average monthly precipitations were classified into four quartiles. This step led to obtaining a dense dataset that is composed of 1404 records, each of which is composed of 9 out of 36 different items. We have conducted different tests. Specifically, we considered a window that contains the transactions for 5, 10 and 20 years, respectively. Similar to previous experiments, for each of these windows, we still considered two support thresholds, namely 10% and 50%. For the size of the incoming slide, we

[5] Data gathered online from the PRISM database at Oregon State University, available at http://www.prism.oregonstate.edu/.

Table 3
Rainfall datasets.

Algorithm	Step (months)	Window (months)	Iterations	Support threshold 50%			Support threshold 10%		
				Diff. (%)	Time (ms)	Memory (KB)	Diff. (%)	Time (ms)	Memory (KB)
Apriori	12	60	112	+44.78%	194	17,432.586	+550.94%	1.035	18,050.746
AWT				+36.57%	183	17,744.984	+542.77%	1.022	18,230.112
MFI				+35.07%	181	9127.380	+154.72%	405	10,882.948
SWF				+32.84%	178	14,726.616	+126.42%	360	18,660.024
WIS					134	10,652.826		159	16,551.826
Apriori	24	120	53	+68.25%	106	17,884.985	+709.52%	850	18,548.018
AWT				+60.32%	101	19,847.571	+585.71%	720	17,483.752
MFI				+49.21%	94	10,299.283	+265.71%	384	13,004.923
SWF				+38.10%	87	12,094.563	+239.05%	356	18,426.018
WIS					63	18,421.760		105	17,436.571
Apriori	48	240	24	+94.19%	167	17,129.120	+990.41%	796	17,790.184
AWT				+46.51%	126	17,112.824	+887.67%	721	17,419.506
MFI				+61.63%	139	11,029.384	+508.22%	444	13,940.069
SWF				+19.77%	103	13,583.968	+313.70%	302	18,090.026
WIS					86	17,784.605		73	16,407.272

considered 1 year (i.e., 12 transactions). A comparison is provided in Table 3. In this test case we also evaluated the performance of the algorithm in terms of the *k-itemsets* that were discovered.

The main advantage of WIS relies on retaining a memory of the candidates that flow across the window. Table 4 shows how many candidate (C) and frequent (F) itemsets enter and quit the related lists in the three cases of the Rainfall dataset. More specifically, we reported the first 5 iterations (after initialization) and in general an average and maximum (in parentheses) number of itemset variations during the whole execution. The minimum value has always been equal to zero and, thus, is not

Table 4
k-Itemset variations of the Rainfall dataset.

	Iteration 1				Iteration 2				Iteration 3				Iteration 4				Iteration 5				Mean (max)			
	C		*F*		*C*		*F*		*C*		*F*		*C*		*F*		*C*		*F*		*C*		*F*	
	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In
(a) *Window* = 60 – *iterations* = 112 – *step* = 12 – *support* = 10%(6)																								
Level 1	1	3	1	3	0	3	0	5	6	0	6	5	1	3	4	5	6	5	6	6	2 (9)	2 (11)	5 (10)	5 (7)
Level 2	18	14	18	14	16	6	22	8	12	0	34	8	13	1	46	8	2	17	40	17	4 (18)	3 (23)	50 (61)	10 (25)
Level 3	37	42	37	42	46	10	40	9	13	0	53	9	5	0	58	9	0	29	49	29	5 (49)	5 (49)	63 (72)	16 (65)
Level 4	27	70	27	70	71	10	23	5	10	0	33	5	0	0	33	5	0	25	28	25	6 (91)	5 (91)	35 (39)	17 (105)
Level 5	8	70	8	70	70	5	4	1	5	0	9	1	0	0	9	1	0	11	8	11	5 (105)	4 (105)	8 (10)	14 (11)
Level 6	1	42	1	42	42	1	0	0	1	0	1	0	0	0	1	0	0	2	1	2	3 (77)	3 (77)	0 (1)	8 (78)
Level 7	0	14	0	14	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 (35)	1 (35)	0 (0)	3 (35)
Level 8	0	2	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (9)	0 (9)	0 (0)	0 (9)
Level 9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (1)	0 (1)	0 (0)	0 (1)
(b) *Window* = 120 – *iterations* = 53 – *step* = 24 – *support* = 10%(12)																								
Level 1	3	0	3	0	2	1	5	1	3	1	7	1	1	4	5	2	1	5	0	1	1 (9)	1 (6)	4 (10)	1 (2)
Level 2	8	0	8	0	0	0	8	0	1	0	9	0	1	4	10	4	6	0	12	0	2 (24)	1 (11)	35 (44)	1 (10)
Level 3	5	0	5	0	0	0	5	0	0	0	5	0	2	6	7	6	10	0	11	0	2 (45)	1 (28)	39 (45)	1 (28)
Level 4	4	0	4	0	0	0	4	0	0	0	4	0	1	4	5	4	8	0	9	0	2 (62)	1 (56)	22 (25)	2 (56)
Level 5	1	0	1	0	0	0	1	0	0	0	1	0	0	1	1	1	2	0	2	0	1 (71)	1 (70)	4 (5)	1 (70)
Level 6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 (56)	1 (56)	0 (10)	1 (56)
Level 7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (28)	0 (28)	0 (0)	0 (28)
Level 8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (8)	0 (8)	0 (0)	0 (8)
Level 9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (1)	0 (1)	0 (0)	0 (1)
(c) *Window* = 240 – *iterations* = 24 – *step* = 24 – *support* = 10%(24)																								
Level 1	1	0	1	0	1	1	2	1	2	2	3	2	5	0	8	2	1	0	8	1	1 (5)	1 (4)	3 (8)	1 (2)
Level 2	13	0	13	0	14	3	27	3	3	0	30	3	4	0	32	1	2	0	33	0	2 (14)	1 (6)	35 (41)	1 (6)
Level 3	30	0	30	0	11	3	41	3	0	0	41	3	3	0	41	0	0	0	41	0	2 (30)	0 (5)	44 (48)	1 (5)
Level 4	25	0	25	0	2	1	27	1	0	0	27	1	1	0	27	0	0	0	27	0	1 (25)	0 (1)	28 (29)	0 (1)
Level 5	8	0	8	0	0	0	8	0	0	0	8	0	0	0	8	0	0	0	8	0	0 (8)	0 (0)	8 (8)	0 (0)
Level 6	1	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0 (1)	0 (0)	1 (1)	0 (0)
Level 7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (0)	0 (0)	0 (0)	0 (0)
Level 8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (0)	0 (0)	0 (0)	0 (0)
Level 9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 (0)	0 (0)	0 (0)	0 (0)

Table 5
MSNBC.com dataset.

Algorithm	Support threshold 50%			Support threshold 10%			Step (month)	Iterations
	Gap (%)	Time (ms)	Memory (KB)	Gap (%)	Time (ms)	Memory (KB)		
Apriori	−673.42%	611	70,373.098	−3300.24%	27.814	62,852.992		
AWT	−637.97%	583	70,576.978	−3207.46%	27.055	69,182.512		
MFI	−408.86%	402	67,004.995	−1468.46%	12.830	58,125.005	1	20
SWF	−329.11%	339	69,043.699	−1201.34%	10.645	69,933.552		
WIS		79	72,588.458		818	72,557.466		
Apriori	−242.54%	459	69,998.760	−1792.94%	23.870	70,931.080		
AWT	−231.34%	444	70,573.032	−1651.94%	22.092	69,729.624		
MFI	−112.69%	285	63,349.501	−688.98%	9.949	57,595.849	10% wind	20
SWF	−2.24%	137	70,445,344	−526,41%	7.899	71,154.200		
WIS		134	70,849,898		1.261	72,288.840		

reported. This test allows us to show the WIS efficiency in preserving candidates from the previous step. This information makes it possible to update the list of frequent item sets efficiently.

For the second benchmark, we chose MSNBC.com Anonymous Web Data dataset, which is available at the UCI Machine Learning Repository. The data come from the Internet Information Server (IIS) logs for msnbc.com and news-related portions of msn.com. Each sequence in the dataset corresponds to page views of a user on 28th September 1999. Each event in the sequence corresponds to page requests, and they are recorded at the level of the page category. To make the stream more consistent, the records were clustered by 10, and duplications were removed. This task led us to obtain a dataset of 98,981 records, with 17 items. The results are reported in Table 5.

We also profiled the WIS execution time at each step compared to Apriori, AWT, MFI and SWF. In this case, we also recorded the initialization time besides the execution time during the first 30 iterations. The dataset used as a benchmark was R10KC20, with a support threshold set at 10%. The execution profiles are reported in Table 6.

Table 6
Execution profiles.

Iteration	R10KC20, Time (ms)				
	Apriori	AWT	MFI	SWF	WIS
Init	1029	1110	577	409	985
1	916	946	432	294	78
2	938	954	431	289	65
3	905	944	429	285	31
4	917	944	439	287	33
5	948	984	452	285	67
6	912	966	443	293	139
7	929	974	441	290	27
8	905	953	434	381	28
9	898	943	436	289	29
10	891	934	446	307	29
11	901	939	445	293	26
12	895	932	430	301	28
13	897	943	435	291	29
14	901	937	433	292	34
15	909	957	431	296	26
16	907	957	433	300	27
17	903	938	441	293	28
18	892	930	440	312	27
19	923	958	444	315	28
20	927	956	444	351	28
21	969	976	450	294	26
22	929	974	444	308	27
23	932	981	446	313	28
24	944	987	449	297	29
25	930	972	439	324	27
26	891	947	456	283	27
27	923	958	445	286	28
28	931	949	460	282	28
29	889	935	436	287	29
30	878	931	443	301	28
TOT	28,459	29,709	13,804	9428	2069

From all of the examples provided above, we can observe that WIS performed faster than MFI and SWF but had a slightly higher memory occupation. This higher occupation arises because WIS retains in memory the whole list of frequent itemsets that were mined in the previous step, while SWF keeps only the 2-itemsets and MFI makes use of transactions that are represented as bit-sequences. Nevertheless, the WIS approach, which is based on a limited test window, can find frequent itemsets faster because the number of candidates to test at each step is lower and related only to those that are involved in the transactions within the test window. In addition, WIS does not require a pass through the dataset to compute the support.

7. Conclusions

In this paper, we considered the problem of mining frequent itemsets in a stream of transactions within a limited window. A variable degree of interest can be expressed on each slot of the window, which usually decreases over time. The traditional approach based on an iterated application of Apriori would incur as a cost from recomputing the whole set of frequent itemsets, even if most of them stay frequent when the window moves ahead. We proposed the Window Itemset Shift (WIS) as an alternative solution, which retains a memory of flowing candidates within a reduced test window. Experimental results, which also included two stream mining algorithms (MFI and SWF), indicate that this approach is advantageous in terms of the execution time, although it is more demanding in terms of the memory occupation. This relationship arises because of the additional data structures, which avoid re-exploring the entire itemset lattice.

References

[1] R. Agrawal, T. Imeilinski, A. Swami, Database mining: a performance perspective, IEEE Trans. Knowl. Data Eng. 5 (6) (1993) 914–925.
[2] R. Agrawal, T. Imeilinski, A. Swami, Mining association rules between sets of items in large databases, ACM SIGMOND Internation Conference Management data, 1993.
[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, 20th VLDB Conference, 1994.
[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A.I. Verkamo, Fast discovery of association rules, Advances in Knowledge Discovery and Data Mining1996.
[5] L. Troiano, G. Scibelli, C. Birtolo, A fast algorithm for mining rare itemsets, Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on, IEEE Computer Society, 2009, pp. 1149–1155.
[6] L. Troiano, L.J. Rodríguez-Muñiz, J. Ranilla, I. Díaz, Interpretability of fuzzy association rules as means of discovering threats to privacy, Int. J. Comput. Math. 89 (2012) 325–333.
[7] I. Díaz, L.J. Rodríguez-Muñiz, L. Troiano, On mining sensitive rules to identify privacy threats, in: J.-S. Pan, M.M. Polycarpou, M. Wozniak, A.C.P.L.F. Carvalho, H. Quintián-Pardo, E. Corchado (Eds.), HAIS, Lecture Notes in Computer Science, vol. 8073, Springer, 2013, pp. 232–241.
[8] L. Troiano, G. Scibelli, A time-efficient breadth-first level-wise lattice-traversal algorithm to discover rare itemsets, Data Min. Knowl. Disc. (2013) 1–35.
[9] L. Szathmary, A. Napoli, P. Valtchev, Towards rare itemset mining, ICTAI'07: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence, IEEE Computer Society, Washington, DC, USA, 2007, pp. 305–312.
[10] G. Yang, The complexity of mining maximal frequent itemsets and maximal frequent patterns, KDD'04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, New York, NY, USA, 2004, pp. 344–353.
[11] In: G. Piatetsky-Shapiro, W.J. Frawley (Eds.), Knowledge Discovery in Databases, AAAI/MIT Press, 1991.
[12] H. Mannila, H. Toivonen, I. Verkamo, Efficient algorithms for discovering association rules, KDD'94: Proceedings of the AAAI Workshop on Knowledge Discovery in Databases, AAAI Press, Seattle, Washington, USA, 1994, pp. 181–192.
[13] A. Savasere, E. Omiecinski, S.B. Navathe, An efficient algorithm for mining association rules in large databases, VLDB'95: Proceedings of the 21th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, pp. 432–444.
[14] S. Brin, R. Motwani, J.D. Ullman, S. Tsur, Dynamic itemset counting and implication rules for market basket data, SIGMOD'97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, ACM, New York, NY, USA, 1997, pp. 255–264.
[15] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, L. Lakhal, Mining frequent patterns with counting inference, SIGKDD Explor. Newsl. 2 (2000) 66–75.
[16] M.J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New algorithms for fast discovery of association rules, Technical Report, 1997, (Rochester, NY, USA).
[17] J. Han, J. Pei, Y. Yin, R. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, in: H. Mannila (Ed.), Data Mining and Knowledge Discovery, Kluwer, New York, NY, USA, 2004, pp. 53–87.
[18] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, D. Yang, H-mine: Hyper-structure mining of frequent patterns in large databases, ICDM'01: Proceedings of the 2001 IEEE International Conference on Data Mining, IEEE Computer Society, Washington, DC, USA, 2001, pp. 441–448.
[19] M. Song, S. Rajasekaran, A transaction mapping algorithm for frequent itemsets mining, IEEE Trans. Knowl. Data Eng. 18 (2006) 472–481.
[20] T. Uno, T. Asai, Y. Uchida, H. Arimura, LCM: an efficient algorithm for enumerating frequent closed item sets, 2003.
[21] T. Uno, M. Kiyomi, H. Arimura, LCM ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets, in: R.J.B. Jr, B. Goethals, M.J. Zaki (Eds.), FIMI, *CEUR Workshop Proceedings*, CEUR-WS.org, vol. 126, 2004.
[22] T. Uno, M. Kiyomi, H. Arimura, LCM ver. 3: collaboration of array, bitmap and prefix tree for frequent itemset mining, Proc. of 1st Int. Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, ACM, NY, USA, 2005, pp. 77–86.
[23] H.-F. Li, S.-Y. Lee, Mining frequent itemsets over data streams using efficient window sliding techniques, Expert Syst. Appl. 36 (2009) 1466–1477.
[24] H.-F. Li, S.-Y. Lee, M.-K. Shan, An efficient algorithm for mining frequent itemsets over the entire history of data streams, 1st Int. Workshop on Knowledge Discovery in Data Streams, in conjunction with 15th European Conference on Machine Learning, Pisa (Italy), 2004.
[25] H.-F. Li, S.-Y. Lee, M.-K. Shan, Online mining (recently) maximal frequent itemsets over data streams, Proceedings of the 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications, RIDE'05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 11–18.
[26] F. Ao, Y. Yan, J. Huang, K. Huang, Mining maximal frequent itemsets in data streams based on fp-tree, Proceedings of the 5th international conference on Machine Learning and Data Mining in Pattern Recognition, MLDM'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 479–489.
[27] Y. Yan, Z. Li, H. Chen, Fast mining maximal frequent itemsets based on fp-tree. In: ER'04, pp. 348–361.
[28] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, Proceedings of the 28th international conference on Very Large Data Bases, VLDB'02, VLDB Endowment, 2002, pp. 346–357.
[29] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD'03, ACM, New York, NY, USA, 2003, pp. 487–492.
[30] C.-H. Lee, C.-R. Lin, M.-S. Chen, Sliding window filtering: an efficient method for incremental mining on a time-variant database, Inf. Syst. 30 (2005) 227–244.
[31] J.H. Chang, W.S. Lee, A sliding window method for finding recently frequent itemsets over online data streams, J. Inf. Sci. Eng. 20 (2004) 753–762.
[32] Y. Chi, H. Wang, P.S. Yu, R.R. Muntz, Catch the moment: maintaining closed frequent itemsets over a data stream sliding window, Knowl. Inf. Syst. 10 (2006) 265–294.

Luigi Troiano Ms.Eng. (2000), Ph.D. (2004). He is Assistant Professor at University of Sannio, and coordinator of Computational and Intelligent System Engineering Lab (CISELab). He obtained Laurea (master degree) in IT Engineering at University of Naples Federico II in 2000, and Ph.D. in IT Engineering at University of Sannio in 2004. His research interests are mainly related to Computational and Intelligent Systems, focusing on how to apply mathematical models and advanced algorithms to Industry.
Giacomo Scibelli Ms.Eng. (2006). He is a Ph.D. student at University of Sannio and currently employed at Poste Italiane S.p.A.. He obtained Laurea (master degree) in IT Engineering at University of Naples Federico II in 2006. His research interests are mainly related to Data Mining and Computational Intelligent Systems with applications to finance.