Need for SPEED: Mining Sequential Patterns in Data Streams

C. Raïssi
EMA-LGI2P/Site EERIE
Parc Scientifique Georges Besse
30035 Nîmes cedex 1 - France
Email: Chedy.Raissi@ema.fr

P. Poncelet
EMA-LGI2P/Site EERIE
Parc Scientifique Georges Besse
30035 Nîmes cedex 1 - France
Email: Pascal.Poncelet@ema.fr

M. Teisseire
LIRMM UMR CNRS 5506
161 Rue Ada
34392 Montpellier cedex 5 - France
Email: teisseire@lirmm.fr

Abstract— **Recently, the data mining community has focused on a new challenging model where data arrives sequentially in the form of continuous rapid streams. It is often referred to as data streams or streaming data. Many real-world applications data are more appropriately handled by the data stream model than by traditional static databases. Such applications can be: stock tickers, network traffic measurements, transaction flows in retail chains, click streams, sensor networks and telecommunications call records. In this paper we propose a new approach, called SPEED (*Sequential Patterns Efficient Extraction in Data streams*), to identify sequential patterns in a data stream. To the best of our knowledge this is the first approach defined for mining sequential patterns in streaming data. The main originality of our mining method is that we use a novel data structure to maintain frequent sequential patterns coupled with a fast pruning strategy. At any time, users can issue requests for frequent sequences over an arbitrary time interval. Furthermore, our approach produces an approximate support answer with an assurance that it will not bypass a user-defined frequency error threshold. Finally the proposed method is analyzed by a series of experiments on different datasets.**

I. INTRODUCTION

Recently, the data mining community has focused on a new challenging model where data arrives sequentially in the form of continuous rapid streams. It is often referred to as data streams or streaming data. Many real-world applications data are more appropriately handled by the data stream model than by traditional static databases. Such applications can be: stock tickers, network traffic measurements, transaction flows in retail chains, click streams, sensor networks and telecommunications call records. In the same way, as the data distribution are usually changing with time, very often end-users are much more interested in the most recent patterns [4]. For example, in network monitoring, changes in the past several minutes of the frequent patterns are useful to detect network intrusions [5].

Due to the large volume of data, data streams can hardly be stored in main memory for on-line processing. A crucial issue in data streaming that has recently attracted significant attention is hence to maintain the most frequent items encountered [8], [9]. For example, algorithms concerned with applications such as answering iceberg query, computing iceberg cubes or identifying large network flows are mainly interested in maintaining frequent items. Furthermore, since data streams are continuous, high-speed and unbounded, it is impossible to mine association rules by using algorithms that require multiple scans. As a consequence new approaches were proposed to maintain itemsets rather than items [12], [6], [4], [10], [22]. Nevertheless, according to the definition of itemsets, they consider that there is no limitation on items order. In this paper we consider that items are really ordered into the streams, therefore we are interested in mining sequences rather than itemsets. To the best of our knowledge, there is no proposition for maintaining such frequent sequences. We propose a new approach, called SPEED (*Sequential Patterns Efficient Extraction in Data streams*), to mine sequential patterns in a data stream. The main originality of our approach is that we use a novel data structure to incrementally maintain frequent sequential patterns (with the help of *tilted-time windows*) coupled with a fast pruning strategy. At any time, users can issue requests for frequent sequences over an arbitrary time interval. Furthermore, our approach produces an approximate support answer with an assurance that it will not bypass a user-defined frequency thresholds.

The remainder of the paper is organized as follows.

Section II goes deeper into presenting the problem statement. In Section III we propose a brief overview of related work and place particular emphasis on sequential patterns mining. The SPEED approach is presented in Section IV. Section V reports the result of our experiments. In Section VI, we summarize our findings and conclude the paper with future avenues for research.

II. PROBLEM STATEMENT

In this section we give the formal definition of the problem of mining sequential patterns in data streams. First, we give a brief overview of the traditional sequence mining problem by summarizing the formal description introduced in [21] and extended in [20]. Second we examine the problem when considering streaming data.

A. Mining of Sequential Patterns

Let DB be a set of customer transactions where each transaction T consists of customer-id, transaction time and a set of items involved in the transaction.
Let $I = \{i_1, i_2...i_m\}$ be a set of literals called items. An itemset is a non-empty set of items. A sequence s is a set of itemsets ordered according to their timestamp. It is denoted by $< s_1 \; s_2 \; ...s_n >$, where $s_j, j \in 1...n$, is an itemset. A k-sequence is a sequence of k items (or of length k). For example, let us consider that a given customer purchased items 1, 2, 3, 4, 5, according to the following sequence: $S =< (1) \; (2, 3) \; (4) \; (5) >$. Therefore, aside from 2 and 3 which were purchased together in a common transaction, all other items in the sequence were bought separately.. s is a 5-sequence.

A sequence $S' =< s'_1 \; s'_2 \; ... \; s'_n >$ is a subsequence of another sequence $S =< s_1 \; s_2 \; ... \; s_m >$, denoted $S' \prec S$, if there exist integers $i_1 < i_2 < ... \; i_j \; ... < i_n$ such that $s'_1 \subseteq s_{i_1}$, $s'_2 \subseteq si_2$, ... $s'_n \subseteq si_n$.

For example, the sequence $S' =< (2) \; (5) >$ is a subsequence of S, i.e. $S' \prec S$ because $(2) \subseteq (2, 3)$ and $(5) \subseteq (5)$. However $< (2) \; (3) >$ is not a subsequence of s since items were not bought during the same transaction.

All transactions from the same customer are grouped together and sorted in increasing order and are called a data sequence. A support value (denoted $supp(S)$) for a sequence gives its number of actual occurrences in DB. Nevertheless, a sequence in a data sequence is taken into account only once to compute the support even if several occurrences are discovered. A data sequence contains a sequence S if S is a subsequence of the data sequence. In order to decide whether a sequence is frequent or not, a minimum support value (denoted *minsupp*) is specified by the user, and the sequence is said to be *frequent* if the condition $supp(S) \geq minsupp$ holds. Given a database of customer transactions the problem of sequential pattern mining is to find all the sequences whose support is greater than a specified threshold (minimum support). Each of these represents a sequential pattern, also called a frequent sequence.
The anti-monotonic Apriori property [1] holds for sequential patterns [18].

B. Sequential Patterns Mining on Data Streams

Let *data stream* $DS = B_{a_i}^{b_i}, B_{a_{i+1}}^{b_{i+1}}, ..., B_{a_n}^{b_n}$ be an infinite sequence of batches, where each batch is associated with a time period $[a_k,b_k]$, i.e. $B_{a_k}^{b_k}$, and let $B_{a_n}^{b_n}$ be the most recent batch. Each batch $B_{a_k}^{b_k}$ consists of a set of customer data sequences; that is, $B_{a_k}^{b_k} = [S_1, S_2, S_3, ..., S_j]$. For each data sequence S in $B_{a_k}^{b_k}$ we are thus provided with its list of itemsets. In the rest of the paper we will consider,without loss of generality, that an itemset is merely reduced to one item. We also assume that batches do not have necessarily the same size. Hence, the length (L) of the data stream is defined as $L = |B_{a_i}^{b_i}| + |B_{a_{i+1}}^{b_{i+1}}| + \ldots + |B_{a_n}^{b_n}|$ where $|B_{a_k}^{b_k}|$ stands for the cardinality of the set $B_{a_k}^{b_k}$.

Batch	Sequence	Items
B_0^1	S_a	(1) (2) (3) (4) (5)
	S_b	(8) (9)
B_1^2	S_c	(1) (2)
B_2^3	S_d	(1) (2) (3)
	S_e	(1) (2) (8) (9)
	S_f	(2) (1)

Fig. 1. The set of batches B_0^1, B_1^2 and B_2^3

In this context, we define the support of a sequential pattern as follows: the support of a sequence S at a specific time interval $[a_i, b_i]$ is denoted by the ratio of the number of customers having sequence S in the current time window to the total number of customers. Therefore, given a user-defined minimum support, the problem of sequential patterns in data streams is to find all frequent patterns S_k over an arbitrary time period $[a_i, b_i]$, i.e. verifying $\sum_{t=a_i}^{b_i} \text{support}_t(S_k) \geq minsupp \times |B_{a_i}^{b_i}|$, of the streaming data using as little main memory as possible.

Example 1: In the rest of the paper we will use this toy example as an illustration, while assuming that the first batch B_0^1 is merely reduced to two customer data sequences. Figure 1 illustrates the set of all batches. Let us now consider the following batch, B_1^2, which only contains one customer data sequence. Finally we will also assume that three customer data sequences are embedded in B_2^3. Let us now assume that the minimum support value is set to 50%. If we look at B_0^1, we obtain the two following maximal frequent patterns: $< (1)(2)(3)(4)(5) >$ and $< (8)(9) >$. If we now consider the time interval $[0-2]$, i.e. batches B_0^1 and B_1^2, maximal frequent patterns are: $< (1)(2) >$. Finally when processing all batches, i.e. a $[0-3]$ time interval, we obtain the following set of frequent patterns: $< (1)(2) >$, $< (1) >$ and $< (2) >$. According to this example, one can notice that the support of the sequences can vary greatly depending on the time periods and so we need to have a framework that enables us to store these time-sensitive supports.

III. Related Work

The task of discovering all the frequent sequences is quite challenging since the search space is extremely large: let $< s_1\ s_2\ ...s_m >$ be a provided sequence and $n_i = |s_j|$ cardinality of an itemset. Then the search space, i.e. the set of all potentially frequent sequences is $2^{n_1+n_2+...n_m}$.

In this section we first propose an overview of traditional approaches used for mining sequential patterns. We then discuss why the approaches are irrelevant in a data stream context. Second, as recent research is interested in considering evolution of databases (*incremental mining*) we will present these approaches in order to measure their relevance to our problem. Finally we propose an overview on recent approaches for data stream mining.

A. Mining sequential patterns

From the definition presented so far, different approaches were proposed to mine sequential patterns. We shall now briefly review the GSP algorithm principle [20] which was the first Apriori-based approach [1]. To build up candidates and frequent sequences, the GSP algorithm makes multiple passes over the database. The first step aims at computing the support of each item in the database. When this step has been completed, the frequent items (i.e. those that satisfy the minimum support) have been discovered. They are considered as frequent 1-sequences (sequences having a single itemset, itself a singleton). The set of candidate 2-sequences is built up according to the following assumption: candidate 2-sequences could be any couple of frequent items, whether embedded in the same transaction or not. Frequent 2-sequences are determined by counting the support. From this point, candidate k-sequences are generated from frequent (k-1)-sequences obtained in pass-(k-1). The main idea of candidate generation is to retrieve, from among (k-1)-sequences, pairs of sequences (S, S') such that discarding the first element of the former and the last element of the latter results in two fully matching sequences. When such a condition holds for a pair (S, S'), a new candidate sequence is built by appending the last item of S' to S. The supports for these candidates are then computed and those with minimum support become frequent sequences. The process iterates until no more candidate sequences are formed. Another method based on the Generating-Pruning principle is PSP [13] where a prefix-tree based approach is used. The methods presented thereafter also use Generating-Pruning approach and need to load the database (or a rewriting of the database) in main memory. For instance, in [23], the SPADE algorithm is proposed and needs only three database scans in order to extract the sequential patterns. SPAM [2] proposes a vertical bitmap representation of the database for both candidate representation and support counting. An original approach for mining sequential patterns aims at recursively projecting the data sequences into smaller databases. Firstly proposed in [7], FREESPAN, and its extension PREFIXSPAN [17], are the first algorithms considering the pattern projection method for mining sequential patterns instead of Generating-Pruning approaches.

Traditional approaches differ from streaming data mining at least in the three following aspects. First, Generating Pruning techniques are irrelevant since the generation is performed through a set of join operations whereas join is a typical blocking operator, i.e. computation for any sequence cannot complete before seeing the past and future data sets [6]. Second, each data element in streaming data should be examined at most once. For instance PREFIXSPAN requires two passes on the database. Finally, memory usage for mining data streams should be bounded even through new data elements are continuously generated from the data stream [12].

B. Considering incremental approaches

As databases evolve, the problem of maintaining sequential patterns over a significantly long period of time becomes essential since a large number of new records may be added to a database. To reflect the current state of the database, in which previous sequential patterns would become irrelevant and new sequential patterns might appear, incremental approaches were proposed. ISE [14] is an efficient algorithm for computing the frequent sequences in the updated database. It minimizes computational costs by re-using the minimal information from the old frequent sequences, the set of candidate sequences to be tested is thus substantially reduced. The SPADE algorithm was extended in the ISM algorithm [16]. In order to update the supports and enumerate frequent sequences, it maintains "maximally frequent sequences" and "minimally infrequent sequences" (i.e. a negative border). KISP [11] also proposes to take advantage of the knowledge previously computed and generates a knowledge base for further queries about sequential patterns of various support values.
Since they are Generating-Pruning based, all these approaches suffer the same drawbacks as traditional approaches. Furthermore, maintaining a border as ISM in data streaming would be very memory consuming and time consuming.

C. Data stream mining approaches

To the best of our knowledge there is no proposition for mining sequential patterns in streaming data. Therefore, in this section we give an overview of approaches for mining all frequent itemsets over the entire history of a streaming data.

The first approach was proposed by [12] where they study the landmark model where patterns support is calculated from the start of the data stream. They also define the first single-pass algorithm for data streams based on the anti-monotonic property. Li et al. [10] use an extended prefix-tree-based representation and a top-down frequent itemset discovery scheme. In [22] they propose a regression-based algorithm to find frequent itemsets in sliding windows. Chi et al. [4] consider closed frequent itemsets and propose the closed enumeration tree (CET) to maintain a dynamically selected set of itemsets.
In [6], authors consider an FP-tree-based algorithm [7] to mine frequent itemsets at multiple time granularities by a novel logarithmic tilted-time window technique. Let us have a closer look at this technique because SPEED also considers tilted-time windows. This notion was first introduced in [3] and is based on the fact that people are often interested in recent changes at a fine granularity but long term changes at a coarse granularity.
In the following, we report example from [6]. Figure 2 shows a natural tilted-time windows table: the most recent 4 quarters of an hour, then ,in another level of granularity, the last 24 hours, and 31 days. Based on this model, one can store and compute data in the last hour with the precision of quarter of an hour, the last day with the precision of hour, and so on. By matching for each sequence of a batch a tilted-time window, we have the flexibility to mine a variety of frequent patterns depending on different time intervals. In [6], the authors propose to extend natural tilted-time windows table to logarithmic tilted-time windows table by simply using a logarithmic time scale as shown in Figure 3. The main advantage is that with one year of data and a finest precision of quarter, this model needs only 17 units of time instead of 35,136 units for the natural model. In order to maintain these tables, the logarithmic tilted-time windows frame will be constructed using different levels of granularity each of them containing a user-defined number of windows. Let $B_1, B_2, \ldots, B_n$ be an infinite sequence of batches where B_1 is the oldest batch. For $i \geq j$, and for a given sequence S, let $f_S(i,j)$ denote the frequency of S in B_j^i where $B_j^i{=}\bigcup_{k=j}^{i} B_k$. By using a logarithmic tilted-time window, the following frequencies of S are kept: $f(n,n)$; $f(n-1,n-1)$; $f(n-2,n-3)$; $f(n-4,n-7)\ldots$. This table is updated as follows. Given a new batch B, we first replace $f(n,n)$, the frequency at the finest level of time granularity (*level 0*), with $f(B)$ and shift back to the next finest level of time granularity (*level 1*). $f(n,n)$ replaces $f(n-1,n-1)$ at level 1. Before shifting $f(n-1,n-1)$ back to level 2, we check if the intermediate window for level 1 is full (in this example the maximum windows for each level is 2). If yes, then $f(n-1,n-1)+f$ is shifted back to level 2. Otherwise it is placed in the intermediate window and the algorithm stops. The process continues until shifting stops. If we received N batches from the stream, the logarithmic tilted-time windows table size will be bounded by $2 \times \lceil log_2(N) \rceil + 2$ which makes this windows schema very space-efficient.

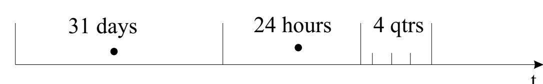


Fig. 2. Natural Tilted-Time Windows Table

According to our problem, all presented approaches

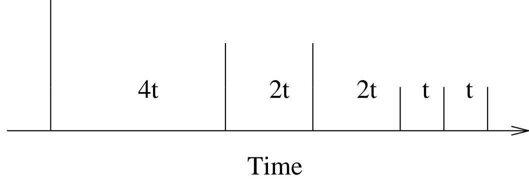


Fig. 3. Logarithmic Tilted-Time Windows Table

consider inter-transaction associations, i.e. there is no limitation on order of events while we consider sequences, which implies a strict order of events. By considering such an order, we are thus provided with a challenging problem since the search space is significantly larger.

IV. THE SPEED APPROACH

In this section we propose the SPEED approach for mining sequential patterns in streaming data.

A. An overview

In this section, we give an overview of the SPEED approach. Our main goal is to mine all maximal sequential patterns over an arbitrary time interval of the stream. The algorithm runs in 2 steps:

- The insertion of each sequence of the studied batch in the data structure $treereg$ using the regions principle.
- The extraction of the maximal subsequences using the LCSP algorithm.

We will now focus on how each new batch is processed then we will have a closer look on the pruning of unfrequent sequences.

Items	Tilted-T W	(Regions, Root$_{Reg}$)
1	$\{[t_0,1]\}$	$\{(1, S_a)\}$
2	$\{[t_0,1]\}$	$\{(1, S_a)\}$
3	$\{[t_0,1]\}$	$\{(1, S_a)\}$
4	$\{[t_0,1]\}$	$\{(1, S_a)\}$
5	$\{[t_0,1]\}$	$\{(1, S_a)\}$

Fig. 4. Updated items after the sequence S_a

Sequences	Size	Tilted-Time Windows
S_a	5	$\{[t_0, 1]\}$
S_b	2	$\{[t_0, 1]\}$

Fig. 5. Updated sequences after the sequence S_b

1) Processing new batches: From the batches from Example 1, our algorithms performs as follows: we process the first sequence S_a in B_0^1 by first storing S_a into our tree ($treereg$). This tree has the following characteristics: each path in $treereg$ is provided with a *region* and sequences in a path are ordered according to the inclusion property. By construction, all subsequences of a sequence are in the same region. This tree is used in order to reduce the search space when comparing and pruning sequences. Furthermore, only "maximal sequences" are stored into $treereg$. These sequences are either sequences directly extracted from batches or their maximal subsequences which are constructed from items in S_a such as all these items are in the same region. Such a merging operation has to respect item order in the sequence, i.e. this order is expressed through their timestamp. By storing only maximal subsequences we aim at storing a minimal number of sequences such that we are able to answer a user query. When the processing of S_a completes, we are provided with a set of items (1..5), one sequence (S_a) and $treereg$ updated. Items are stored as illustrated in Figure 4. The "*Tilted-T W*" attribute is the number of occurrences of the corresponding item in the batch. The "$Root_{reg}$" attribute stands for the root of the corresponding region in $treereg$. Of course, for one region we only have one $Root_{Reg}$ and we also can have several regions for one item. For sequences (C.f. Figure 5), we store both the size of the sequence and the associated tilted-time window. This information will be useful during the pruning phase. The left part of the Figure 6 illustrates how the $treereg$ tree is updated when considering S_a.

Let us now process the second sequence of B_0^1. Since S_b is not a subsequence of S_a, it is inserted in $treereg$ in a new valuation (C.f. subtree S_b in Figure 6).

Items	Tilted-T W	(Regions, Root$_{Reg}$)
1	$\{[t_0,1], [t_1,1]\}$	$\{(1, S_a)\}$
2	$\{[t_0,1], [t_1,1]\}$	$\{(1, S_a)\}$
...	...	... ...
8	$\{[t_0,1]\}$	$\{(2, S_b)\}$
9	$\{[t_0,1]\}$	$\{(2, S_b)\}$

Fig. 7. Updated items after B_1^1

Let us now consider the batch B_1^2 merely reduced to S_c. Since items 1 and 2 already exist in the set of sequences, their tilted-time windows must be updated (C.f. Figure 7). Furthermore, items 1 and 2 are in the same region: 1 and the longest subsequence for these

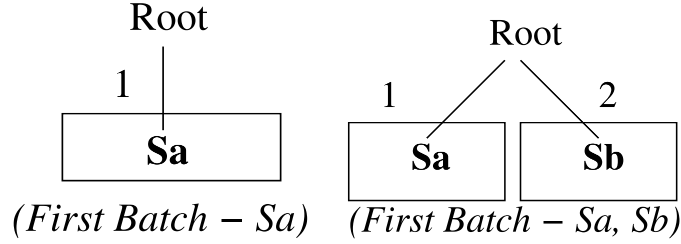


Fig. 6. The region tree after the first batch

Sequences	Size	Tilted-Time Windows
S_a	5	$\{[t_0, 1]\}$
S_c	2	$\{[t_0, 1], [t_1, 1], [t_2, 1]\}$
S_d	3	$\{[t_0, 1], [t_2, 1]\}$
...	...	...

Fig. 8. Updated sequences after S_d of B_2^2

items is $<(1)\ (2)>$, i.e. S_c which is included in S_a. We thus have to insert S_c in $treereg$ in the region 1. Nevertheless as S_c is a subsequence of S_a that means that when S_a occurs in previous batch it also occurs for S_c. So the tilted-time window of S_c has to be also updated.

Items	Tilted-T W	(Regions, Root$_{Reg}$)
1	$\{[t_0,1], [t_1,1], [t_2,2]\}$	$\{(1, S_a)\}$ $\{(2, S_e)\}$ $\{(3, S_f)\}$
2	$\{[t_0,1], [t_1,1], [t_2,2]\}$	$\{(1, S_a)\}$ $\{(2, S_e)\}$ $\{(3, S_f)\}$
...	...	...

Fig. 10. Updated items after the sequence S_f

Sequences	Size	Tilted-Time Windows
S_a	5	$\{[t_0, 1]\}$
S_b	2	$\{[t_0, 1], [t_2, 1]\}$
S_c	2	$\{[t_0, 1], [t_1, 1], [t_2, 2]\}$
S_d	3	$\{[t_0, 1], [t_2, 1]\}$
S_e	4	$\{[t_2, 1]\}$
S_f	2	$\{[t_2, 1]\}$

Fig. 11. Updated sequences after S_f of B_2^3

The sequence S_d is considered in the same way as S_c (C.f. Figure 9 and Figure 11). Let us now have a closer look on the sequence S_e. We can notice that items 1 and 2 are in region 1 while items 8 and 9 are in region 2. We can believe that we are provided with a new region. Nevertheless, we can notice that in fact the sequence $<(8)(9)>$ already exist in $treereg$ and is a subsequence of S_e. The longest subsequence of S_e in the region 1 is $<(1)(2)>$. In the same way, the longest subsequence of S_e for region 2 is $<(8)(9)>$. As we are provided with two different regions and $<(8)(9)>$ is a root of region for 2, we do not create a new region but we insert S_e as a root of region for 2 and we insert the subsequence $<(1)(2)>$ both on tree for region 1 and 2. Of course, tilted-time windows are updated. Finally we proceed to the last sequence S_f. We can notice that the order between itemsets is different from previous sequences. When parsing the set of items, we can conclude that they occur in the same region 1. Nevertheless the longest subsequences are reduced to $<(1)>$ and $<(2)>$, i.e. neither $S_f \prec S_c$ or $S_c \prec S_f$ holds, then we have to consider a new region.

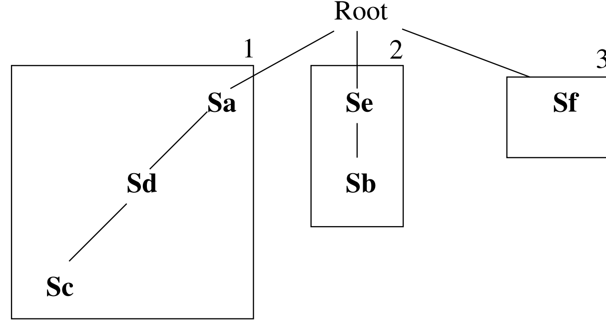
2) Pruning sequences: Let us now discuss how stored sequences are pruned. While pruning in [6] is done in 2 distinct operations, SPEED prunes unfrequent patterns in a single operation which is in fact a dropping of the tail sequences of tilted-time windows $f_S(t_k), f_S(t_{k+1}), \ldots, f_S(t_n)$ when the following condition holds:

$$\forall i, k \leq i \leq n, support_{a_i}^{b_i}(S) < \varepsilon_f |B_{a_i}^{b_i}|$$

By navigating into $treereg$, and by using the regions index, we can directly and rapidly prune irrelevant sequences without further computations. This process is repeated after each new batch in order to use as little main memory as possible. During the pruning phase, titled-windows are merged in the same way as in [6].

B. The SPEED algorithm

We describe in more detail the SPEED algorithm (C.f. Algorithm 1). While batches are available, we consider sequences embedded in batches in order to update



(Third Batch – Se)

Fig. 9. The valuation tree after batches processing

Algorithm 1: The SPEED algorithm

Data: an infinite set of batches B=B_0^0, B_1^1, ... B_n^n ...; a $minsupp$ user-defined threshold; an error rate ϵ.

Result: A set of frequent items and sequences

// init phase

$treereg \leftarrow \emptyset$; $ITEMS \leftarrow \emptyset$; $SEQS \leftarrow \emptyset$;

$region \leftarrow 1$;

while *batches are available* **do**

  foreach $B_i^i \in B$ **do**

    UPDATE(B_i^i, $treereg$, $ITEMS$, $SEQS$, $minsupp$, ε);

    PRUNE($treereg$, $ITEMS$, $SEQS$, $minsupp$, ε);

---

our structures (UPDATE). Then we prune unfrequent sequences in order to maintain our structures in main memory (PRUNETREE). In the following, we consider that we are provided with the three next structures. Each value of $ITEMS$ is a tuple ($labelitem$, $\{time, occ\}$, $\{(regions$, $Root_{Reg})\}$) where $labelitem$ stands for the considered item, $\{time, occ\}$ is used in order to store the number of occurrences of the item for different time of batches and for each region in $\{regions\}$ we store its associated sequences ($Root_{Reg}$ in the $treereg$ structure. According to the following property, the number of regions is limited.

Property 1: Let Φ be the number of items in DS. The maximal number of regions is bounded by $\Phi^2 + 1$.

Proof: Let Φ be the number of items. We can generate Φ^2 maximal sequences of size 2 and one maximal sequence. Each of them stands for a region. Whatever the added sequence, it will be a subsequence and will be included in one of the already existing $\Phi^2 + 1$ regions.

In other words, in the worst case, our algorithm has to check, for each sequence embedded in a batch $\Phi^2 + 1$ regions.

The $SEQS$ structure is used to store sequences. Each value of $SEQS$ is a tuple (s, $size(s)$,$\{time, occ\}$) where $size(s)$ stands for the number of items embedded in s. Finally, the $treereg$ structure is a tree where each node is a sequence stored in $SEQS$ and where vertices correspond to the associated region (according to the previous overview).

Let us now examine the Update algorithm (C.f. Algorithm 2) which is the main core of our approach. We consider each sequence embedded in the batch. From a sequence S, we first get regions of all its items (GETREGIONS). If items were not already considered we only have to insert S in a new region. Otherwise, we extract all different regions associated on items of S. For each region the GETFIRSTSEQOFVAL function returns a new subsequence S_x constructed by merging items sharing same region with their associated $Root_{Reg}$. We then compute the longest common subsequences of S_x in $Root_{Val}$ by applying the LCSP *Longest Common Sequential Patterns* function. This function returns an empty set both when there are no subsequences or if subsequences are merely reduced to one item[1].

Property 2: Let u, v be two sequences and $|u|$, $|v|$

[1]Due to lack of space, we do not describe this function. Interested reader may refer to [19]. LCSP is an extension of the NKY algorithm [15] of time complexity *O(n(m - r))* where *n* and *m* are the sizes of sequences and *r* the size of the longest maximal sequence.

Algorithm 2: The UPDATE algorithm

Data: a batch $B_t^t = [S_1, S_2, S_3, ..., S_k]$; a $minsupp$ user-defined threshold; an error rate ϵ.
Result: $treereg$, $ITEMS$, $SEQS$ updated.

foreach $sequence\ Seq \in B_t^t$ **do**
- $LatticeMerge \leftarrow \emptyset$; $DelayedInsert \leftarrow \emptyset$;
- $Candidates \leftarrow$ GETREGIONS(Seq);
- **if** $Candidates = \emptyset$ **then**
  - INSERT(Seq,$NewVal++$);
- **else**
  - **foreach** $region\ Val \in Candidates$ **do**
    - // Get the maximal sequence from region Val
    - $FirstSeq \leftarrow$ GETFIRSTSEQOFVAL(Val);
    - // Compute all the longest common
    - // subsequences
    - $NewSeq \leftarrow$ LCSP(Seq,$FirstSeq$);
    - **if** $|NewSeq| = 1$ **then**
      - // There is a direct inclusion
      - // between the two tested sequences
      - **if** $(NewSeq[0] == Seq) || (NewSeq[0] == FirstSeq)$ **then**
        - $LatticeMerge \leftarrow Val$;
      - **else**
        - // Found a new subsequence
        - // to be added
        - INSERT($NewSeq[0], Val$);
        - UPDATETTW($NewSeq[0]$);
        - $DelayedInsert \leftarrow NewSeq$;
    - **else**
      - $DelayedInsert \leftarrow Seq$;
      - **foreach** $sequence\ S \in NewSeq$ **do**
        - INSERT(S, Val);
        - UPDATETTW(S);
        - $DelayedInsert \leftarrow S$;
  - // Create a new region
  - **if** $|LatticeMerge| = 0$ **then**
    - INSERT($Seq, NewVal++$);
    - UPDATETTW(Seq);
  - **else**
    - **if** $|LatticeMerge| = 1$ **then**
      - INSERT($Seq, LatticeMerge[0]$);
      - UPDATETTW(Seq);
    - **else**
      - MERGE($LatticeMerge, Seq$);
    - INSERTANDUPDATEALL($DelayedInsert$, $LatticeMerge[0]$);

the associated size. Let r be the size of the maximal subsequence between u and v. Let Λ be the number of maximal subsequences. We have: $\Lambda \leq \binom{w=min(|u|,|v|)}{r}$.
Proof: Let u and v be two sequences. We can obtain respectively $2^{|u|}$ and $2^{|v|}$ subsequences. The set of maximal subsequences having size r is then: $min(\binom{|u|}{r}, \binom{|v|}{r}) \equiv \binom{w=min(|u|,|v|)}{r}$

If there is only one subsequence, i.e. cardinality of NewSeq is 1, we know that the subsequence is either a root of region or S_x itself. We thus store it in a temporary array (*LatticeMerge*). This array will be used in order to avoid to create a new region if it already exists a root of region included in S. Otherwise we know that we are provided with a subsequence and then we insert it into $treereg$ (INSERT) and propagate the tilted-time window (UPDATETTW). Sequences are also stored in a temporary array (*DelayedInsert*). If there exist more than one subsequence, then we insert all these subsequences on the corresponding region and also store with S on *DelayedInsert* them in order to delay their insertion for a new region. If *LatticeMerge* is empty we know that it does not exist any subsequence of S included on sequences of $treereg$ and then we can directly insert S in a new region. Otherwise, we insert the subsequence in $treereg$ for the region of $LatticeMerge$. If the cardinality of $LatticeMerge$ is greater than 1, we are provided with a sequence which will be a new root of region and then we insert it. For the both last case, we insert all the set of subsequences embedded and we update their tilted-time windows (INSERTANDUPDATEALL).

Property 3: Let $treereg$ be the structure at the end of the process. Let S and S' be two sequences such as $S' \preceq S$, then:

1) If S' does not exist in $treereg$ then S also does not exist in $treereg$.
2) If S' exists in $treereg$, let $Sup'_0, \ldots, Sup'_n$ (resp. $Sup_0, \ldots, Sup_m$ for S) be the supports of S'[2] in all of its tilted-time windows then:
$n \geq m$ and $Sup'_i \geq Sup_i$,$\forall i$ such as $0 \leq i \leq m$

Proof:

1) The first part is proved by induction on N, i.e. the number of batches.
For $N = 0$ (the oldest batch), if $S \in treereg$ then $Support_0(S) \geq \epsilon_f |B_0^0|$. Let us consider that $S' \notin treereg$, as $S' \not\subseteq S$ we have $Support_0(S) \leq$

[2]where Sup'_0 is the support in the most recent window.

$\epsilon_f|B_0^0|$. So we have $S \notin treereg$.
For $N > 0$, let us consider $S' \notin treereg$ after processing the Nth batch. We have to consider the two following cases: (i) $S' \notin treereg$ after the processing of the $(N-1)^{th}$ batch then by induction we also have $S \notin treereg$ by extending in the same way the case $N = 0$. (ii) $S' \in treereg$ after the $(N-1)^{th}$ batch and the sequence was pruned when processing the batch N. As $S' \not\subseteq S$, it exists $S_1, \ldots, S_p$ such as $S_1 = S'$ and $S_p = S$, where S_i is the subsequence of S_{i+1} for $1 \leq i \leq p$. With the pruning condition, we know that $S_1, \ldots, S_p$ were already pruned during the processing of the batch N thus $S \notin treereg$.

2) As $S' \leq S$, we have $S_1, \ldots, S_p$ (cf previous part). By using the pruning condition, we know that the table of tilted-time windows of S_i has much more windows than S_{i+1} with $1 \leq i \leq p$, thus S' has much more windows that S $(m \geq n)$.
By definition, we know that $Sup_0, \ldots, Sup_{n-1}$ and $Sup'_0, \ldots, Sup'_{n-1}$ are the support of S and S' for each batch. These windows have the same structure, we can thus apply the anti-monotonic property: $Supp_i(S') \geq Supp_i(S)$ for $1 \leq i \leq n-1$.
Let us assume W_s (resp. $W_{s'}$), the set of sequences having incremented S (resp. S') for the batch N, i.e $Supp_n(S) = |W_s|$. We thus have $W_s \subseteq W_{s'}$ and by the antimonotonic property: $Supp_q(S') = |W_{s'}| \geq Supp_q(S) = |W_s|$.

Maintaining all the data streams in the main memory requires too much space. So we have to store only relevant sequences and drop sequences when the tail-dropping condition holds. When all the tilted-time windows of the sequence are dropped the entire sequence is dropped from $treereg$. As a result of the tail-dropping we no longer have an exact frequency over L, rather an approximate frequency. Now let us denote $F_S(L)$ the frequency of the sequence in all batches and $\tilde{F}_S(L)$ the approximate frequency. With $\varepsilon \ll minsupp$ this approximation is assured to be less than the actual frequency according to the following inequality [6]: $F_S(L) - \epsilon|L| \leq \tilde{F}_S(L) \leq F_S(L)$.

Due to lack of space we do not present the entire PRUNE algorithm we rather explain how it performs. First all sequences verifying the pruning constraint are stored in a temporary set ($ToPrune$). We then consider items in $ITEMS$ data structure. If an item is unfrequent, then we navigate through $treereg$ in order: (i) to prune this item in sequences; (ii) to prune sequences in $treereg$ also appearing in $ToPrune$. This function takes advantage of the anti-monotonic property as well as the order of stored sequences. It performs as follows, nodes in $treereg$, i.e. sequences, are pruned until a node occurring in the path and having siblings is found. Otherwise, each sequence is updated by pruning the unfrequent item. When an item remains frequent, we only have to prune sequences in $ToPrune$ by navigating into $treereg$.

V. EXPERIMENTS

In this section, we report our experiments results. We describe our experimental procedures and then our results.

A. Experimental Procedures

The stream data was generated by the IBM synthetic market-basket data generator, available at http://www.almaden.ibm.com/cs/quest. In all the experiments we used 1K distinct items and generated 1M of transactions. Furthermore, we have fixed minsupp at 10%. We conducted two sets of experiments, in the first, we set the frequency error threshold at 0.1 with an average sequence length of 3 or 5 itemsets and in the second we set the frequency error threshold at 0.2 with the same sequence lengths. The stream was broken in batches of 20 seconds for the 3-sequences and 90 seconds for the 5-sequences. Furthermore, all the transactions can be fed to our program through standard input. Finally, our algorithm was written in C++ and compiled using gcc without any optimizations flags. All the experiments were performed on an AMD Athlon XP-2200 running Linux with 512 MB of RAM.

B. Results

At each processing of a batch the following informations were collected: the size of the SPEED data structure at the end of each batch in bytes, the total number of seconds required per batch, the total number of maximal sequences generated for this batch and the number of valuations present on the data stream sequences. The x axis represents the batch number.

Figure 12 show time results for 3 and 5-sequences. Every two batches the algorithm needs more time to process sequences, this is in fact due to the merge operation of the tilted-time windows which is done in our experiments every 2 batches on the finest granularity level. The jump in the algorithm is thus the result of extra computation cycles needed to merge the tilted-time windows values for all the nodes in the $treereg$ structure.

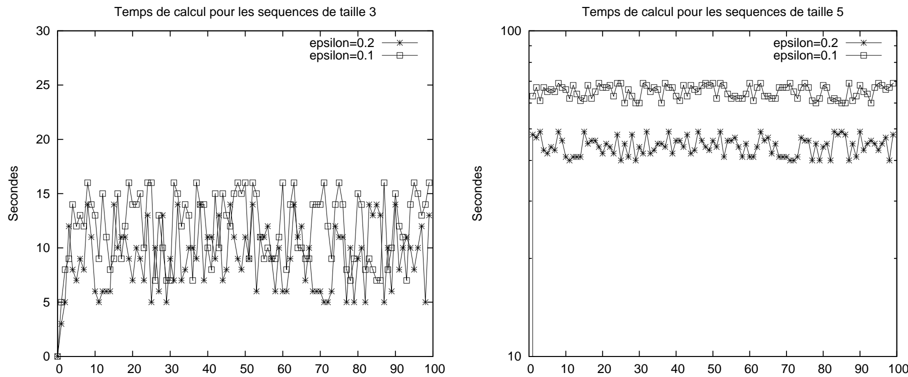


Fig. 12. Speed Time requirements for $\{3,5\}$-sequences

The time requirements of the algorithm tend to grow very slowly as the stream progresses and do not excess the 20 or the 90 seconds computation time limit for every batch. Figure 13 show memory needs for the processing of our sequences. Space requirements is bounded for 3-sequences by 35M and 78M for the 5-sequences, this requirement is however acceptable as this can easily fit in main memory. Experiments show that the SPEED algorithm can handle sequences in data streams without falling behind the stream as long as we choose correct batch duration values. However, development is not over yet and we still feel that improvements and optimizations can still be made to the current implementation.

VI. CONCLUSION

In this paper we addressed the problem of mining sequential patterns in streaming data and proposed the first approach, called SPEED, for mining such patterns. SPEED is based on a new efficient structure and on strict valuation of edges. Such a valuation is very useful either when considering the pruning phase or when comparing sequences since we only have to consider sequences embedded into the tree sharing same valuations. Thanks to the anti-monotonic property and the order of stored sequences in our structure, the pruning phase is also improved. Conducted experiments have shown that our approach is efficient for mining sequential patterns in data stream. Furthermore, with SPEED, users can, at any time, issue requests for frequent sequences over an arbitrary time interval.

There are various avenues for future work. First, we are investigating how the tilted-time windows could be improved in order to store both the time interval and the distribution of items during a period. With such a functionality we expect that the approximation of the returned result from a request with a large time interval could be improved. Secondly, we are currently studying how long sequences, i.e. sequences embedded in different consecutive batches, can be considered in SPEED. Finally we would like to develop query answering techniques adapted to our approach.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large database. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD 93)*, pages 207–216, 1993.

[2] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using bitmap representation. In *Proceedings of the 8th SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 02)*, pages 439–435, Alberta, Canada, 2002.

[3] Y. Chen, G. Dong, J. Han, B. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB 02)*, pages 322–334, Hong Kong, China, 2002.

[4] Y. Chi, H. Wang, P.S. Yu, and R.R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 04)*, pages 59–66, Brighton, UK, 2004.

[5] P. Dokas, L. Ertoz, V. Kumar, A. Lazarevic, J. Srivastava, and P.-N. Tan. Data mining for network intrusion detection. In *Proceedings of the 2002 National Science Foundation Workshop on Data Mining*, pages 21–30, 2002.

[6] G. Giannella, J. Han, J. Pei, X. Yan, and P. Yu. Mining frequent patterns in data streams at multiple time granularities. In *In H. Kargupta, A. Joshi, K. Sivakumar and Y. Yesha (Eds.), Next Generation Data Mining, MIT Press*, 2003.

[7] J. Han, J. Pei, B. Mortazavi-asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD 00)*, pages 355–359, Boston, USA, 2000.

[8] C. Jin, W. Qian, C. Sha, J.-X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings*

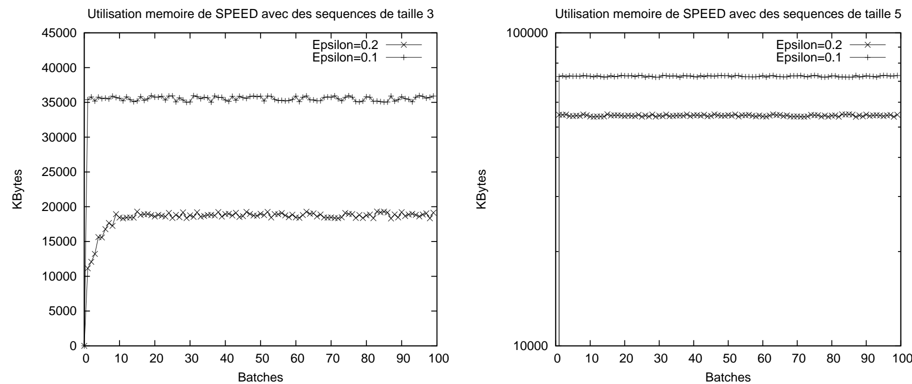


Fig. 13. Speed Memory requirements for {3-5}-sequences

of the 12th International Conference on Information and Knowledge Management (CIKM 04), pages 287–294, New Orleans, Louisiana, 2003.

[9] R.-M. Karp, S. Shenker, and C.-H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[10] H.-F. Li, S.Y. Lee, and M.-K. Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proceedings of the 1st International Workshop on Knowledge Discovery in Data Streams*, Pisa, Italy, 2004.

[11] M. Lin and S. Lee. Improving the efficiency of interactive sequential pattern mining by incremental pattern discovery. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences - CDROM*, Big Island, USA, 2003.

[12] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 02)*, pages 346–357, Hong Kong, China, 2002.

[13] F. Masseglia, F. Cathala, and P. Poncelet. The PSP approach for mining sequential patterns. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD 98)*, pages 176–184, Nantes, France, 1998.

[14] F. Masseglia, P. Poncelet, and M. Teisseire. Incremental mining of sequential patterns in large databases. *Data and Knowledge Engineering*, 46(1):97–121, 2003.

[15] Yajima Shuzo Nakatsu Narao, Kambayashi Yahiko. A longest common subsequence suitable for similar text strings. *Acta Informatica*, 18(1):171–179, 1982.

[16] S. Parthasarathy, M. Zaki, M. Orihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Proceedings of the 8th International Conference on Information and Knowledge Management (CIKM 99)*, pages 251–258, Kansas City, MO, USA, 1999.

[17] J. Pei, J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, and U. Dayal. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of 17th International Conference on Data Engineering (ICDE 01)*, pages 215–224, Heidelberg, Germany, 2001.

[18] J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. In *Proceedings of the 10th International Conference on Information and Knowledge Management (CIKM 02)*, pages 18–25, MCLean, USA, 2002.

[19] C. Raïssi. Mining sequential patterns on data streams (in french). Master's thesis, University Montpellier II, June 2005.

[20] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT 96)*, pages 3–17, Avignon, France, 1996.

[21] R. Agrawal R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE 95)*, pages 3–14, Tapei, Taiwan, 1995.

[22] W.-G. Teng, M.-S. Chen, and P.S. Yu. A regression-based temporal patterns mining schema for data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 03)*, pages 93–104, Berlin, Germany, 2003.

[23] M.J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 42(1):31–60, February 2001.