Reservoir Pattern Sampling in Data Streams

Arnaud Giacometti, Arnaud Soulet

Reservoir Pattern Sampling in Data Streams

Arnaud Giacometti[0000−0003−0270−5146] and Arnaud Soulet[0000−0001−8335−6069]✉

Université de Tours, LIFAT, Blois, France
firstname.lastname@univ-tours.fr

Abstract. Many applications generate data streams where online analysis needs are essential. In this context, pattern mining is a complex task because it requires access to all data observations. To overcome this problem, the state-of-the-art methods maintain a data sample or a compact data structure retaining only recent information on the main patterns. This paper addresses online pattern discovery in data streams based on pattern sampling techniques. Benefiting from reservoir sampling, we propose a generic algorithm, named ResPat, that uses a limited memory space and that integrates a wide spectrum of temporal biases simulating landmark window, sliding window or exponential damped window. For these three window models, we provide fast damping optimizations and we study their temporal complexity. Experiments show that the performance of ResPat algorithms is particularly good. Finally, we illustrate the interest of our approach with online outlier detection in data streams.

1 Introduction

Many applications generate data streams, especially with the rise of network sensors [2] and the Internet of Things [8]. Beyond their operational utility, the analysis of these data streams raise strategic issue in mobile data stream mining [25], online transaction analysis [18] and so on. In most cases, it is not possible to consider storing these data to perform an off-line analysis because of their volume. In addition, the usefulness of certain analyzes like early outlier detection necessarily relies on online processing. Unfortunately, knowledge discovery in data streams remains a challenging task due to the continuous arrival of data observations that must be processed in a short time (*time constraint*) despite a limited memory space (*space constraint*) [19,24]. This problem is particularly exacerbated for pattern mining whose combinatorial complexity is costly by nature [11,16]. It aims to maintain a collection of interesting patterns extracted from the data stream while respecting these constraints. For this purpose, a first strategy consists in maintaining a compact data structure containing the information on the pattern occurrences appearing in each data observation [15,21,22,26]. In addition to being expensive to update, the size of this structure is not limited and sometimes requires significant memory space. A second strategy is to maintain a data sample representative of the data stream. It is then possible to mine the interesting patterns from this sample [1,4,23]. However, the expensive cost

of this mining step prevents it from being repeated at the arrival of each data observation and then, from having an up-to-date collection of patterns.

This paper revisits the pattern discovery in data streams at the light of *pattern sampling* [3,5]. Frequent pattern sampling consists in drawing patterns at random proportionally to their frequency. In our context, the principle is to maintain a sample of k patterns representative of the data stream. For example, a pattern twice as frequent will be twice as likely to be picked. To the best of our knowledge, no method exists to sample patterns in data streams. For this purpose, the key idea is to benefit from *reservoir sampling*. This family of randomized algorithms picks a random sample of k items from a population of unknown size in a single pass over the items [10,20,27]. Unfortunately, the existing reservoir sampling methods are not suitable to deal with two challenges: output space and temporal bias. First, existing reservoir sampling methods are designed to perform input space sampling (i.e., from data observations) and not output space sampling (i.e., from patterns covering the data observations). Of course, it is not possible to enumerate the exponential number of patterns contained in each data observation to build the population to be sampled. Second, there are reservoir sampling methods that take into account a static distribution on the population [10]. To the best of our knowledge, existing reservoir sampling methods do not incorporate a temporal bias to favor the most recent observations. However, the damping of the oldest data observations is important for pattern mining in data streams [17].

This paper provides the first pattern sampling method in data streams using reservoir sampling. The general principle is to generate a key for each occurrence of patterns and to keep in the reservoir the k occurrences with the largest keys. More specifically, our contributions are as follows:

- We present a generic algorithm ResPat that performs exponential random jumps in the output space so as not to compute a key for each occurrence and that updates the value of the keys of the reservoir to integrate several window models. We also propose *fast damping* optimized algorithms ($\text{ResPat}_{\text{no}}$, $\text{ResPat}_{\text{win}}$ and $\text{ResPat}_{\text{exp}}$) for three window models that avoid having to explicitly modify the keys of the reservoir.
- We demonstrate that the proposed ResPat algorithm family based on reservoir sampling is exact and that it requires a memory space linear with the sample size k. Interestingly, our theoretical study proves its efficiency by computing the complexity of the number of insertions in the reservoir.
- We evaluate the effectiveness of our algorithm family by performing experiments on UCI benchmarks and synthetic data. This experimental study shows the important contribution of the exponential jump and fast damping optimizations. A use case also illustrates the interest of pattern sampling to detect outliers in data streams. In particular, our online and one-pass method rediscovers the outliers of an off-line method with a good accuracy.

The outline of this paper is as follows. Section 2 reviews some related work about pattern mining in data streams and pattern sampling methods. Section 3 introduces basic definitions and the formal problem statement. We present our

reservoir pattern sampling algorithms for data streams in Section 4. We evaluate our approach in Section 5 and conclude in Section 6.

2 Related Work

Data mining over data streams is a daunting task [19,24], especially pattern mining [11,16]. Most of the existing methods aim to extract all the frequent patterns and more rarely, are limited to the top-k frequent patterns [28] or other measure like max-frequency [6]. Itemsets is the most popular pattern language and only few works are interested in particular forms like maximal patterns [18] or closed patterns [7,22]. Several static window models [17] are implemented to consider (i) the entire stream from a certain time (landmark window) [7,21,28], (ii) only the data observations inside a window (sliding window) [7,22,26] or (iii) the entire data stream by weighting the observations to favor the most recent ones (damped window) [15,23]. Clearly this latter is the more complex model and it is also the least dealt with in the literature. The majority of methods relies on a tree-like data structure in order to efficiently store and manipulate the current mined patterns [15,21,22,26]. This structure is updated according to the data stream to maintain a collection taking into account the considered window. Statistical techniques such as the Chernoff bound [28] are often used to estimate the frequency of the patterns in order to safely remove the less promising ones. Most of these techniques compute approximated collection of patterns contrary to our proposal, which guarantees an exact sampling: whatever the window model, the mined sample is equivalent to what would be mined if all the data observations were stored in memory.

Rather than incrementally maintaining the collection of interesting patterns, another approach consists in incrementally maintaining a data sample representative of the data stream benefiting from reservoir sampling [10,20,27]. The idea is then to extract the collection of patterns from this data sample by simulating different window models (e.g., sliding window [4], exponential bias [1] or tilted window [23]). Unfortunately, this approach makes it necessary to repeat the pattern discovery after each modification of the data sample, which is very costly (both for the frequent pattern mining and for the pattern sampling). For this reason, it would be more advantageous to directly sample the output space (i.e., pattern space) instead of the input space (i.e., data space). To the best of our knowledge, there are methods for sequential data [9] but not for sampling patterns in a data stream. First, stochastic methods [3] require evaluating the measure m on the entire data for selecting the next state of the random walk. This evaluation is impossible in a data stream because we do not have all the data observations. Second, multi-step random procedures [5,9] have the advantage of not directly evaluating the measure. They consist in drawing a data observation proportionally to the utility sum of patterns that it contains and then, in drawing a pattern from these patterns proportionally to its utility. Unfortunately, the essential normalization constant for drawing the transaction will only be known at the end of the stream. In short, all the pattern sampling methods require a

	Data stream $\mathcal{D}$					Damping function ω		
Time	Time.	Items				ω_{no}	ω^2_{win}	$\omega^{0.3}_{\text{exp}}$
↓	0	A	B		D	1	0	0.223
	1	A	B	C	D	1	0	0.301
	2	A		C	E	1	0	0.406
	3	A	B	C		1	1	0.549
	4			C	D E	1	1	0.741
	5			C	D E	1	1	1.000

Table 1. A running example with three damping functions

full-access to data incompatible with the notion of data stream where not all the data observations can be stored. Thus, this work proposes to extend reservoir sampling methods dedicated to the input space so that they effectively deal with the output space with a time bias.

3 Preliminaries

Data stream and patterns This paper exclusively addresses itemset language. Given a set of literals $\mathcal{I}$, a data stream is a sequence of transactions with timestamps: $\mathcal{D} = \langle (t_1, d_1), \ldots, (t_n, d_n) \rangle$ such that $d_j \subseteq \mathcal{I}$ for $j \in [1..n]$ and $t_j < t_{j+1}$ for $j \in [1..n-1]$. Without loss of generality, we consider that $t_1 = 0$. The itemset language $\mathcal{L}_{\mathcal{I}} = 2^{\mathcal{I}}$ is the set of all patterns (or itemsets). The cover of a pattern φ in a data stream $\mathcal{D}$, denoted by $\mathcal{D}[\varphi]$, is the set of data observations containing φ: $\mathcal{D}[\varphi] = \{(t, d) \in \mathcal{D} \ : \ \varphi \subseteq d\}$. For instance, Table 1 provides a data stream with 6 data observations described by 5 items $\mathcal{I} = \{A, B, C, D, E\}$. ABD is the transaction of the first data observation containing 8 itemsets: $2^{d_0} = \{\emptyset, A, B, D, AB, AD, BD, ABD\}$. Of course, a data stream evolves over the time with the addition of new data observations (e.g., a transaction will likely be added at timestamp 6).

Interestingness measure A damping function $\omega : \Re^+ \to [0, 1]$ is a decreasing function that assigns a lower weight to the oldest data observations such that $\omega(0) = 1$. This damping function enables us to consider the different existing window models [17]: **(i) Landmark window:** A landmark window ω_{no} considers all the data observations equally since a landmark point (as t_1 without loss of generality): $\omega_{\text{no}} : t \mapsto 1$ **(ii) Sliding window:** The (time-stamp based) sliding window ω^T_{win} considers only the most recent data observations [12]. Formally, $\omega^T_{\text{win}} : t \mapsto 1$ if $t \leq T$ and 0 otherwise, where $T > 0$ is the window time size. **(iii) Exponential damped window:** A popular damped window is the exponential bias [1] defined as $\omega^\alpha_{\text{exp}} : t \mapsto e^{-\alpha t}$ where α is the damping factor. For instance, Table 1 illustrates this three damping functions. It is easy to see that the damping functions ω^2_{win} and $\omega^{0.3}_{\text{exp}}$ gives a larger weight to the most recent observations. This is suitable for applications where people are interested only in the most

recent information of the data streams. For this purpose, we weight the support with the damping function:

Definition 1 (Damping support). *Given a damping function ω, the damped support of a pattern φ in $\mathcal{D} = \langle (t_1, d_1), \ldots, (t_n, d_n) \rangle$ is defined as below:*

$$supp_\omega(\varphi, \mathcal{D}) = \frac{\sum_{(t,d) \in \mathcal{D}[\varphi]} \omega(t_n - t)}{\sum_{(t,d) \in \mathcal{D}} \omega(t_n - t)}$$

Obviously, the damping function ω_{no} leads to the traditional support. For instance, $supp_{\omega_{\text{no}}}(AB, \mathcal{D}) = 3/6$ and $supp_{\omega_{\text{no}}}(CDE, \mathcal{D}) = 2/6$ meaning that AB occurs more frequently than CDE. On the latest data observations, the situation is reversed: $supp_{\omega^2_{\text{win}}}(AB, \mathcal{D}) = 1/3$ (or $supp_{\omega^{0.3}_{\text{exp}}}(AB, \mathcal{D}) = 1.073/3.220$) is lower than $supp_{\omega^2_{\text{win}}}(CDE, \mathcal{D}) = 2/3$ (or $supp_{\omega^{0.3}_{\text{exp}}}(CDE, \mathcal{D}) = 1.741/3.220$).

Problem Statement Let Ω be a population and $f : \Omega \to [0, 1]$ be a measure, the notation $x \sim f(\Omega)$ means that the element x is drawn randomly from Ω with a probability distribution $\pi(x) = f(x)/Z$ where Z is a normalizing constant.

Given a data stream $\mathcal{D} = \langle (t_1, d_1), \ldots, (t_n, d_n) \rangle$, a language $\mathcal{L}_\mathcal{I}$ and a damping function ω, we aim at selecting k patterns $\varphi_1, \ldots, \varphi_k$ in $\mathcal{L}_\mathcal{I}$ where the probability of each pattern φ_i to be selected is determined by its relative weight $supp_\omega(\varphi_i, \mathcal{D})$: $\varphi_i \sim supp_\omega(\mathcal{L}_\mathcal{I}, \mathcal{D})$ for $i \in \{1, \ldots, k\}$.

As mentioned in the introduction, a method for processing data streams must comply with two constraints: **(i) Space constraint:** In most cases, it is not possible to store all the observations in the data stream $\mathcal{D}$. Therefore, the sampling method has to be done in a single pass to avoid disk storage and the space complexity has to be independent of the number of observations n. **(ii) Time constraint:** Each observation must be processed in a short time to avoid the accumulation of continuously arriving data, which would violate the above space constraint. Next sections address this problem using reservoir sampling.

4 Reservoir Algorithms for Pattern Sampling

4.1 Challenges and key ideas

First, we reformulate our *pattern* sampling problem as an *occurrence* sampling problem. Drawing a pattern according to the weighted support $supp_\omega$ is equivalent to drawing an occurrence according to the damping function ω: $\varphi \sim supp_\omega(\mathcal{L}_\mathcal{I}, \mathcal{D}) \Leftrightarrow \varphi \sim \omega(\mathcal{L}(\mathcal{D}))$ where the multi-set $\mathcal{L}(\mathcal{D}) = \bigcup_{(t,d) \in \mathcal{D}} 2^d$ gathers all the occurrences of patterns from $\mathcal{D}$:

$$\mathcal{L}(\mathcal{D}) = \{ \underbrace{\varphi_1^0, \varphi_1^1, \ldots}_{2^{d_1} \text{ with } \omega(t_n - t_1)} , \underbrace{\varphi_2^0, \varphi_2^1, \ldots}_{2^{d_2} \text{ with } \omega(t_n - t_2)} , \ldots , \underbrace{\varphi_n^0, \varphi_n^1, \ldots}_{2^{d_n} \text{ with } \omega(t_n - t_n)} \}$$

and each occurrence $\varphi_j^i \subseteq d_j$ has a weight $\omega(\varphi_j^i) = \omega(t_n - t_j)$. Interestingly, this reformulation of the problem makes it possible to directly reuse reservoir sampling algorithms from the literature where the occurrences form the population

to be sampled. More precisely, this family of algorithms selects a sample (often without replacement) of a population having an unknown size in a single pass. Some algorithms having the best complexity rely on a central result that we also use in the rest of this paper. Given two keys $key_1 = {u_1}^{1/\omega_1}$ and $key_2 = {u_2}^{1/\omega_2}$ where $\omega_i > 0$ and u_i is uniformly drawn from $[0, 1]$, we have:

$$P(key_1 \leq key_2) = \frac{\omega_2}{\omega_1 + \omega_2} \tag{1}$$

Based on this observation, [10] demonstrates that assigning each occurrence in $\mathcal{L}(\mathcal{D})$ a key ${u_i}^{1/\omega_i}$ where u_i is a random number and then, selecting the k patterns with the largest keys is equivalent to sampling k occurrences *without* replacement from $\mathcal{L}(\mathcal{D})$ proportionally to ω. Besides, as the number of occurrences is very large (i.e., $|\mathcal{L}(\mathcal{D})| \gg k$), a sampling method *without* replacement is equivalent to a sampling method *with* replacement. Consequently, this paper benefits from some principles of the sampling algorithm without replacement proposed by [10].

The context of pattern sampling raises two challenges with respect to the use of reservoir sampling. The first challenge is to address the output space $\mathcal{L}(\mathcal{D})$ rather than the input space $\mathcal{D}$. We could naively apply a reservoir sampling method by enumerating the output space (i.e., all the occurrences for each data observation). For the itemset language, this approach would lead to an exponential complexity $2^{|d|}$ for processing a data observation d, which the time constraint prevents. *Inserting step* in Section 4.2 shows how to avoid the enumeration of the output space by directly selecting the occurrence to insert into the sample. The second challenge is to take into account the damping function ω in the maintenance of the sample. In Equation 1, the weights are static, while in our context, they are dynamic. At the insertion time t_{ins}, all occurrences have 1 as weight – by definition of the damping function ω, we have $\omega(t_{ins} - t_{ins}) = \omega(0) = 1$. When new observations arrive, the weight of the patterns in the sample decreases except for the landmark window. Whatever the damping function, *Damping step* in the next subsection shows how to integrate this modification relying on Equation 1. Finally, Section 4.3 proposes fast damping optimizations for ω^T_{win} and $\omega^\alpha_{\text{exp}}$.

4.2 Generic algorithm: ResPat

Overview This section presents our generic algorithm to address the two above challenges. Algorithm 1 takes a data stream $\mathcal{D}$, a damping function ω and a sample size k as inputs and returns a sample $\mathcal{S}$ containing k patterns randomly drawn in $\mathcal{L}_\mathcal{I}$ proportionally to the damped support in $\mathcal{D}$. Its general principle is to process each observation one by one. The inserting step (lines 7 to 11) inserts some occurrences of the jth observation without enumerating all the output space. The damping step (lines 5 and 6) modifies the keys of the occurrences contained in the sample to integrate the damping function ω.

Before detailing the two main steps, it is important to note that the reservoir $\mathcal{S}$ is a set of triples $\langle key, \varphi, t\rangle$ meaning that the occurrence φ was inserted at time t with the key key. The function MinKey (lines 13-17) returns the smallest key

Algorithm 1 ResPat: Pattern sampling in data streams with damping

Require: A data stream $\mathcal{D}$, a damping function ω and a number of patterns k
Ensure: A sample $\mathcal{S}$ containing k patterns randomly drawn in $\mathcal{L}_{\mathcal{I}}$ proportionally to the damped support in $\mathcal{D}$

1: $\mathcal{S} := \emptyset$
2: $jump := 0$
3: **for** $j \in \langle 1, \ldots, n \rangle$ **do**
4: $\quad i := 0$
// Damping step
5: $\quad$ **if** $j \geq 2$ **then**
6: $\quad\quad$ Update the key of each element $\langle key, \varphi, t \rangle \in \mathcal{S}$ with $key^{\omega(t_{j-1}-t)/\omega(t_j-t)}$
// Inserting step
7: $\quad$ **while** $i + jump < |2^{d_j}|$ **do**
8: $\quad\quad i := i + jump$
9: $\quad\quad$ UpdateSample$(\mathcal{S}, k, unif(\text{MinKey}(\mathcal{S}, k), 1), \mathcal{L}(d_j)^i, t_j)$
10: $\quad\quad jump := \lfloor \log(unif(0,1)) / \log(\text{MinKey}(\mathcal{S},\text{k})) \rfloor + 1$
11: $\quad jump := (i + jump) - |2^{d_j}|$
12: **return** the sample $\mathcal{S}$
13: **function** MinKey$(\mathcal{S}, k)$ // Return the minimum key in $\mathcal{S}$
14: $\quad$ **if** $|\mathcal{S}| < k$ **then**
15: $\quad\quad$ **return** 0
16: $\quad$ **else**
17: $\quad\quad$ **return** $\min_{\langle key, \varphi, t \rangle \in \mathcal{S}} key$
18: **procedure** UpdateSample$(\mathcal{S}, k, key, \varphi, t)$ // Add the pattern φ in $\mathcal{S}$
19: $\quad e := \langle key, \varphi, t \rangle$
20: $\quad$ **if** $|\mathcal{S}| < k$ **then**
21: $\quad\quad$ Add the pattern e in $\mathcal{S}$
22: $\quad$ **else**
23: $\quad\quad$ Replace the pattern with the minimum key in $\mathcal{S}$ by the pattern e

of the sample if the sample contains k occurrences and 0 otherwise. The function UpdateSample (lines 18-23) inserts an occurrence into the sample and removes the occurrence with the smallest key (if necessary for maintaining $|\mathcal{S}| \leq k$). Note that the first k occurrences at the beginning of the stream are automatically added into the sample due to the smallest key that equals to zero (lines 20-21).

Inserting step A naive algorithm could draw a uniform number between 0 and 1, saying u, for each occurrence and it could insert this occurrence when $u^{1/\omega} = u$ exceeds the smallest key m (because the weight of each occurrence is 1). Instead of enumerating one by one all the occurrences, it is enough to calculate how many occurrences, saying $jump$, should be drawn before having the one that will be inserted in the reservoir [20]. Intuitively, to insert a pattern in the reservoir, it is sufficient to calculate the weight ω so that the key $u^{1/\omega}$ is larger than the smallest key in the reservoir, saying m. As each occurrence has a weight of 1, this weight ω simply corresponds to the number of occurrences skipped and the

equation to solve is $m \leq u^{1/jump}$. This intuition is formalized by the below property:

Property 1 (Exponential random jump [20]). Given the smallest key m, the number of occurrences to jump before reaching the occurrence to be inserted in the sample is given by the random variable X_m:

$$X_m = \left\lfloor \frac{\log unif(0,1)}{\log m} \right\rfloor + 1$$

Property 1 is the first key ingredient of the inserting step at line 10 of Algorithm 1. Let us assume in our running example provided by Table 1 that after having processed the first two transactions, the smallest key of the reservoir is $m = 0.8$. If we draw $u = 0.1$, then we get $jump = \left\lfloor \frac{\log 0.1}{\log 0.8} \right\rfloor + 1 = 11$ and the 11th occurrence of $ACDE$ is inserted into the reservoir by replacing the occurrence having the smallest key m. With this technique, there is 1 single draw instead of 11 with a naive enumeration. We will measure this significant gain both theoretically (in Section 4.4) and practically (in Section 5).

For the random jump to be really useful, we must access the $jump$-th occurrence without listing all the previous ones. For this purpose, we introduce the notion of index operator. Given a data observation $(t, d) \in \mathcal{D}$, an *index operator* is a bijection mapping each number $i \in [0..|2^d| - 1]$ to a pattern $\varphi \in 2^d$:

Definition 2 (Itemset index operator). *Given a transaction $d = \{I_0, \ldots, I_{|d|-1}\}$ and an index number $i \in [0..2^{|d|} - 1]$, we consider its value $b_{|d|-1} \ldots b_1 b_0$ in binary system and then, the itemset X returned by $\mathcal{L}(d)^i$ contains all items I_j where $b_j = 1$: $I_j \in X \Leftrightarrow b_j = 1$.*

This index operator is the second key ingredient used at line 9 of Algorithm 1 for selecting directly the right occurrence to insert in the sample without enumerating all the patterns and applying a filter. For instance, $\mathcal{L}(ACDE)^{11}$ is ACE because the decimal value $(11)_{10}$ has $(1011)_2$ as binary value. This operator is efficient because its complexity is linear with the number of items in d.

Damping step All patterns are inserted into the reservoir with 1 as initial weight but after, their weight must be decreased to take into account the damping function ω. In other words, at every time t_j, a pattern inserted at t_{ins} must have a key $u^{1/\omega(t_j - t_{ins})}$. For this purpose, the key of each pattern is raised to the power $\frac{\omega(t_{j-1} - t_{ins})}{\omega(t_j - t_{ins})}$ at each iteration j. The below property formalizes this idea:

Property 2 (Key damping). Given a pattern φ inserted at time t_{ins} with a key key considering a damping function ω, we have for any $t_j \geq t_{ins}$:

$$key^{\frac{\omega(t_{ins} - t_{ins})}{\omega(t_{ins+1} - t_{ins}))} \frac{\omega(t_{ins+1} - t_{ins})}{\omega(t_{ins+2} - t_{ins})} \cdots \frac{\omega(t_{j-1} - t_{ins})}{\omega(t_j - t_{ins})}} = key^{1/\omega(t_j - t_{ins})}$$

Due to lack of space, the proofs are omitted. This property follows from the fact that the left hand-side of the equality can be rewritten as $key^{\prod_{i=ins+1}^{j} \frac{\omega(t_{i-1} - t_{ins})}{\omega(t_i - t_{ins})}}$ and the simplification of the exponent gives

$key^{1/\omega(t_j - t_{ins})}$. Assume that ACE was inserted with the initial key $key = 0.9$ at time 2. Of course, this key will not be modified with the damping function ω_{no}. For the function ω^2_{win}, it decreases to 0 with the sixth observation $(5, CDE)$ because $key^{\omega^2_{\text{win}}(4-2)/\omega^2_{\text{win}}(5-2)} = key^{1/0}$ is equal to 0 (by convention). Finally, with the function $\omega^{0.3}_{\text{exp}}$, this weight is $0.9^{\omega^{0.3}_{\text{exp}}(2-2)/\omega^{0.3}_{\text{exp}}(3-2)} = 0.867$ at time 3, $0.867^{\omega^{0.3}_{\text{exp}}(3-2)/\omega^{0.3}_{\text{exp}}(4-2)} = 0.825$ at time 4 and $0.825^{\omega^{0.3}_{\text{exp}}(4-2)/\omega^{0.3}_{\text{exp}}(5-2)} = 0.772$ at time 5 that also corresponds to $0.9^{1/\omega^{0.3}_{\text{exp}}(5-2)} = 0.772$ as desired.

It is clear that the damping step decreases all the keys (except for ω_{no}) and therefore, the smallest key m in the sample. Therefore, the stronger the damping, the smaller the size of the jumps at the inserting step (see Property 1). Consequently, our approach is less efficient with the damping functions focusing on the latest data observations. Furthermore, the computational cost of decreasing the keys is an important defect of this damping step because this operation is done for each element in $\mathcal{S}$ at every iteration t_j. Fortunately, it is sometimes possible to skip this step for some damping functions as shown in the next section.

4.3 Fast damping algorithms: ResPat$_{\text{no}}$, ResPat$_{\text{win}}$ and ResPat$_{\text{exp}}$

This section improves the generic algorithm ResPat by emulating the key damping without explicitly updating the key of each element contained in the reservoir. Of course, the damping step is useless for ω_{no} as the key value is not modified (indeed, $key^{\omega_{\text{no}}(t_{j-1}-t)/\omega_{\text{no}}(t_j-t)} = key^{1/1} = key$). Consequently, lines 5 and 6 can be safely removed leading to the algorithm denoted by ResPat$_{\text{no}}$. But, we show below that it is also possible to remove these lines for ω^T_{win} and $\omega^\alpha_{\text{exp}}$ by adapting the functions MinKey and UpdateSample.

Sliding window Ideally, the patterns that are too old (i.e., $|t - t_{ins}| > T$) should be removed from the sample $\mathcal{S}$ at each damping step. Our key idea is to count them as being missing from $\mathcal{S}$ even if we do not take them out. Consequently, the number of patterns contained in the sample takes into account the too old patterns (see lines 2 and 8 of Algorithm 2). In practice, this number is maintained at each modification of $\mathcal{S}$ (lines 9 and 11) (with a circular array storing the number of insertions for the last T times). In particular, if the pattern in $\mathcal{S}$ with the minimum key (line 11) is too old, then this pattern is removed and the pattern with the next minimum key is considered. For instance, assume that ACE was inserted at time 2 with the initial key $key = 0.9$ and it remains in the reservoir $\mathcal{S}$ until the end of time 4. At the beginning of time 5, the function MinKey will return 0 whatever the value of the smallest key in $\mathcal{S}$ because ACE is expired (due to $5 - 2 \geq 3$). The insertion of a new pattern will refill the reservoir with the correct number of unexpired patterns. Later, by taking care of timestamps, when the smallest key will be that of an expired itemset (here, ACE with 0.9), this element will be removed from $\mathcal{S}$ and the next smallest key will be considered.

Exponential damping Whatever the insertion time of the key, we can observe that the damping function $\omega^\alpha_{\text{exp}}$ raises the key to the same power (because $\omega^\alpha_{\text{exp}}$ is a memory-less bias function [1]). The below property formalizes this intuition:

Algorithm 2 ResPat$_{\text{win}}$: ResPat with sliding window fast damping

1: **function** MinKey-Win($\mathcal{S}, k, t$) // Return the minimum key in $\mathcal{S}$
2: **if** $|\{\langle key, \varphi, u\rangle \in \mathcal{S} \ : \ |t-u| \leq T\}| < k$ **then**
3: **return** 0
4: **else**
5: **return** $\min_{\langle key,\varphi,u\rangle \in \mathcal{S} \text{ s.t. } |t-u|\leq T} key$
6: **procedure** UpdateSample-Win($\mathcal{S}, k, key, \varphi, t$) // Add the pattern φ in $\mathcal{S}$
7: $e := \langle key, \varphi, t\rangle$
8: **if** $|\{\langle key, \varphi, u\rangle \in \mathcal{S} \ : \ |t-u| \leq T\}| < k$ **then**
9: Add the pattern e in $\mathcal{S}$
10: **else**
11: Replace the pattern with the minimum key in $\mathcal{S}$ by the pattern e

Property 3 (Exponential key damping). Given a pattern φ inserted at time t_{ins} with a key key and an exponent α, we have for any $t_j \geq t_{ins}$:

$$key^{1/\exp(\alpha \times t_{ins})^{\exp(\alpha \times t_j)}} = key^{1/\omega^{\alpha}_{\exp}(t_j - t_{ins})}$$

This property follows from the fact that the two exponents can be rewritten as a single one $\frac{\exp(\alpha \times t_j)}{\exp(\alpha \times t_{ins})} = 1/\exp(-\alpha \times (t_j - t_{ins}))$. Property 3 means that the insertion time t_{ins} is useless for damping the key (if the initial insertion weight takes it into account). For instance, let us take again the example of the itemset ACE inserted at time 2 with the initial key $key = 0.9$. Benefiting from the exponential key damping for $\omega^{0.3}_{\exp}$, this itemset is inserted with the weight $key^{1/\exp(0.3\times 2)} = 0.944$ and its damped key at time 5 is $0.944^{\exp(0.3\times 5)} = 0.772$ (that equals to $0.9^{1/\omega^{0.3}_{\exp}(5-2)} = 0.9^{\exp(-0.3(5-2))}$ as desired). More generally, ResPat$_{\text{exp}}$ benefits from this damping strategy (see Algorithm 3). At line 7, we insert a new pattern at time t_{ins} with a weight greater than 1 so that the weight of the patterns already within the reservoir do not change. Then, when we consult the minimum key at line 5, we correct all the weights with the same power using Property 3.

Algorithm 3 ResPat$_{\text{exp}}$: ResPat with exponential fast damping

1: **function** MinKey-Exp($\mathcal{S}, k, t$) // Return the minimum key in $\mathcal{S}$
2: **if** $|\mathcal{S}| < k$ **then**
3: **return** 0
4: **else**
5: **return** $\min_{\langle key,\varphi,t\rangle \in \mathcal{S}} key^{\exp(\alpha \times t)}$
6: **procedure** UpdateSample-Exp($\mathcal{S}, k, key, \varphi, t$) // Add the pattern φ in $\mathcal{S}$
7: $e := \langle key^{1/\exp(\alpha \times t)}, \varphi, t\rangle$
8: **if** $|\mathcal{S}| < k$ **then**
9: Add the pattern e in $\mathcal{S}$
10: **else**
11: Replace the pattern with the minimum key in $\mathcal{S}$ by the pattern e

4.4 Theoretical analysis

This section studies the ResPat algorithm family. First, the following property proves that these algorithms return a sample with the expected characteristics:

Property 4 (Correctness). Considering that $|\mathcal{L}(\mathcal{D})| \gg k$, the generic algorithm ResPat and its fast damping variants ($\text{ResPat}_{\text{no}}$, $\text{ResPat}_{\text{win}}$ and $\text{ResPat}_{\text{exp}}$) are correct: Given a data stream $\mathcal{D}$, a language $\mathcal{L}_{\mathcal{I}}$ and a damping function ω, each algorithm returns k patterns $\varphi_1, \ldots, \varphi_k$ such that $\varphi_i \sim supp_{\omega}(\mathcal{L}_{\mathcal{I}}, \mathcal{D})$.

This non-trivial property relies on [10] by proving that the temporal bias is correctly maintained thanks to the different key raising properties.

We now analyze the complexity of the algorithm family ResPat. First, the space complexity of the different algorithms is linear with the sample size k. To the best of our knowledge, our proposal has the smallest space complexity for frequent pattern sampling and it is the first one-pass algorithm for frequent pattern sampling. Second, considering the time complexity, it is clear that the efficiency of fast damping algorithms depends essentially on the number of insertions (especially with the optimized versions without damping step). Of course, the lower this number, the more efficient the approach.

Property 5 (Number of insertions). Given a data stream $\mathcal{D}$, a language $\mathcal{L}_{\mathcal{I}}$ and a sample size k, the expected number of insertions into the reservoir is (after the filling phase): (i) $O(k \cdot \log\left(\frac{|\mathcal{L}(\mathcal{D})|}{k}\right))$ for the landmark window ω_{no} (see [10]), (ii) $O(k/T \cdot n)$ for the sliding window ω^{T}_{win} ($T \ll n$) and (iii) $O(k \cdot \alpha \cdot n)$ for the damped window $\omega^{\alpha}_{\text{exp}}$

Unlike other pattern methods in data streams [7,22,26], the weaker the damping is, the more efficient the approach is. In particular, for the sliding window, the larger the window T, the lower the number of insertions. For the exponential damping, the lower the exponent α, the lower the number of insertions.

5 Experimental Evaluation

This experimental study investigates the performance of our reservoir sampling approach in Section 5.1 and it shows its interest for outlier detection in Section 5.2. We use 12 benchmark datasets coming from the UCI Machine Learning repository and the FIMI repository, and 3 large synthetic datasets coming from [21]. Note that the condition $|\mathcal{L}(\mathcal{D})| \gg k$ is satisfied by these large datasets: $|\mathcal{L}(\mathcal{D})| \gg k$. The methods are implemented with the Java language[1]. All experiments are performed on a 2.5 GHz Xeon processor with the Linux operating system and 2 GB of RAM memory. Each of the reported measurements is the mean of 10 runs where the transactions were swapped.

	No damping ω_{no}			Sliding window $\omega_{\text{win}}^{1000}$			Exp. damping $\omega_{\text{exp}}^{0.003}$		
Dataset	2-STEP	A-RES	RESPAT$_{\text{no}}$	A-RES	RESPAT	RESPAT$_{\text{win}}$	A-RES	RESPAT	RESPAT$_{\text{exp}}$
abalone	0.079	0.8	0.8	75.9	81.1	2.1	79.5	84.4	2.9
chess	0.220	-	10.5	-	102.1	28.4	-	83.7	14.3
cmc	0.080	0.8	0.7	25.2	26.2	1.0	27.7	28.6	1.5
connect	0.725	-	14.3	-	2433.0	746.9	-	1569.5	20.0
crx	0.121	4.0	2.0	18.1	13.9	2.1	20.6	16.1	2.5
hypo	0.132	61.2	3.5	146.5	73.1	8.0	170.5	71.2	6.0
mushroom	0.201	-	5.6	-	218.3	30.8	-	184.9	14.3
retail *	6.669	-	115.9	-	2260.2	1025.6	-	1651.1	116.8
sick	0.153	851.8	4.4	938.4	71.7	10.4	964.2	65.1	7.3
T10I4D100K *	0.730	-	7.6	-	1948.3	334.0	-	2104.5	83.3
T10I4D1000K *	oom	-	5.2	-	17835.9	1615.2	-	19624.4	73.1
T15I6D1000K *	oom	-	9.1	-	19395.0	3367.6	-	19387.3	18.2
T30I20D1000K *	oom	-	6474.0	-	-	21229.8	-	29903.9	7008.3
vehicle	0.124	22.9	3.0	46.7	18.2	3.2	51.8	21.2	3.7
waveform	0.165	955.6	4.6	1104.3	131.0	17.6	1144.6	114.6	9.8

oom: out of memory / -: out of time ($\geq$ 10h) / *: variable size transactions

Table 2. Running time in seconds for sampling 100k patterns

5.1 Global and longitudinal performance study

The first experiment assesses the overall efficiency by measuring its total execution time. For this purpose, Table 2 compares the RESPAT algorithm family with two state-of-the-art algorithms: the two step random procedure 2-STEP [5] and the baseline A-RES [10] that corresponds to RESPAT without the exponential random jump (see Property 1). We consider the execution times for a sample size k =100,000 and three damping functions: no damping ω_{no}, a sliding window $\omega_{\text{win}}^{1000}$ and an exponential damping $\omega_{\text{exp}}^{0.003}$. First, we observe that our RESPAT algorithm family manages to process large datasets with 1000K transactions, while the two step random procedure 2-STEP does not have enough memory (denoted by oom in Table 2). Of course, in return, our reservoir sampling approach is slower than 2-STEP. Second, it is clear that the fast damping algorithms RESPAT$_{\text{no}}$, RESPAT$_{\text{win}}$ and RESPAT$_{\text{exp}}$ outperform the baseline A-RES and the generic algorithm RESPAT. Indeed, as soon as the number of items per transaction is large, the exponential random jump becomes mandatory to maintain a reasonable processing time explaining timeouts for A-RES (denoted by -). Besides, the cost of the generic damping step is really prohibitive when the sample size is large because of the high number of key decreases for RESPAT. It is more efficient to simulate these key decreases as for the sliding window (see RESPAT$_{\text{win}}$ column) as well for the exponential damping (see RESPAT$_{\text{exp}}$ column). Finally, Figure 1 plots the execution times of RESPAT$_{\text{no}}$, RESPAT$_{\text{win}}$ and RESPAT$_{\text{exp}}$ for the 6 largest datasets with respect to the sample size k. Unfortunately, our approach struggles for datasets containing very long transactions (here, retail and T30I20D1000K) regardless of the sample size k. Indeed, for very long transactions, the random jump is no longer enough to curb the combinatorial explosion of

[1] The source code is available: https://github.com/asoulet/ResPat

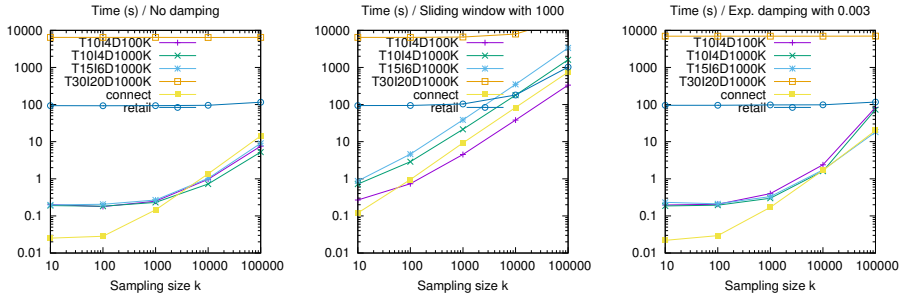

Fig. 1. Running time in seconds (y-axis) with respect to the sample size (x-axis)

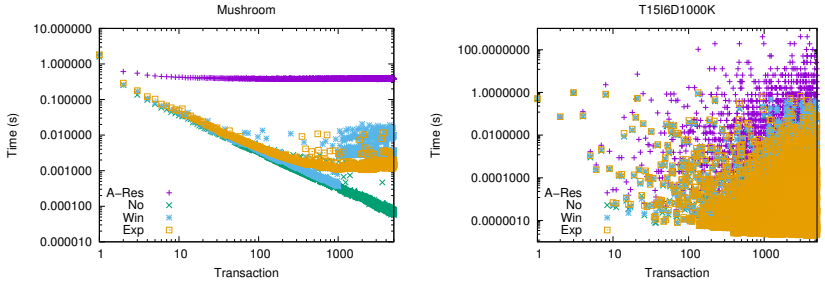

Fig. 2. Longitudinal performance of reservoir sampling algorithms

the occurrence space. In contrast, for the other four datasets, the asymptotically linear behavior is visible in accordance with Property 5.

This second experiment assesses the longitudinal efficiency of our algorithm by measuring the execution time that is required for processing each transaction. Figure 2 plots the execution time per transaction for a sample size $k = 100,000$ and four algorithms: `A-Res`/ResPat$_{\text{no}}$ for ω_{no}, ResPat$_{\text{win}}$ for $\omega_{\text{win}}^{1000}$ and ResPat$_{\text{exp}}$ for $\omega_{\text{exp}}^{0.003}$. We consider two datasets `mushroom` and `T15I6D1000K` that respectively represent the datasets with fixed size transactions and the datasets with variable size transactions (denoted by * in Table 2). First, we see again the strong impact of the exponential random jump that drastically reduces the execution time per transaction once the first transactions have been completed (magenta dots are above the others). Second, for `mushroom`, the execution time decreases steadily when there is no damping ω_{no}. For $\omega_{\text{win}}^{1000}$, a disturbance is observed once the sliding window moves (after 1000) with values varying between a few milliseconds and several tens of milliseconds. For the exponential damping $\omega_{\text{exp}}^{0.003}$, the execution time per transaction stabilizes around a few milliseconds. Third, for `T15I6D1000K`, the situation is less visible because the execution time for each transaction depends strongly on its size. Because of the logarithmic

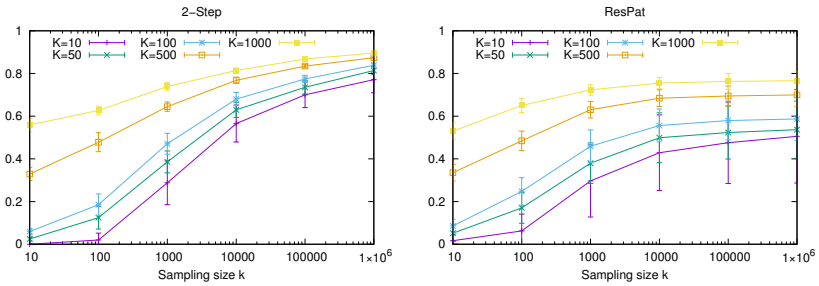

Fig. 3. Accuracy comparison between 2-STEP (left) and RESPAT (right)

scale and high dot density, one might imagine that the execution time increases, which is not true on average.

5.2 Use case: One-pass frequent pattern outlier detection

This section illustrates the interest of our sampling technique to detect outliers by performing a single pass on the data (as it is necessarily the case in a stream). We aim to rediscover the K outliers that we would have obtained with a multi-pass FPOF method. More precisely, the frequent pattern outlier factor of a transaction $d \in \mathcal{D}$ is defined as: $FPOF(d,\mathcal{D}) = \frac{\sum_{\varphi \subseteq d} supp(\varphi,\mathcal{D})}{\max_{d' \in \mathcal{D}} \sum_{\varphi \subseteq d'} supp(\varphi,\mathcal{D})}$. The lower this score is, the more likely to be an outlier the transaction is. Therefore, our goal is to detect the K transactions minimizing this score. We benefit from the formula proposed in [14] to approximate the FPOF from a sample of patterns $\mathcal{S}$ (drawn from $\mathcal{D}$ with respect to the support):

$$\lim_{|\mathcal{S}|=\infty} \underbrace{\frac{|\{\varphi \in \mathcal{S} \ : \ \varphi \subseteq d\}|}{\max_{d' \in \mathcal{D}} |\{\varphi \in \mathcal{S} \ : \ \varphi \subseteq d'\}|}}_{FPOF(d,\mathcal{S})} = FPOF(d,\mathcal{D})$$

Basically, the idea is to maintain a sample of frequent patterns $\mathcal{S}$ with our reservoir sampling approach. At the same time, we apply this formula using the current sample to estimate the FPOF of each transaction. The K transactions minimizing the FPOF are kept throughout the pass on the dataset. At the end, the remaining transactions are considered to be the K outliers. Of course, the sample computed on the first transactions is not very representative of the entire dataset (i.e., far from the final sample) and it is possible to miss true outliers.

Figure 3 plots the average accuracy of a multi-pass FPOF (2-STEP) and a one-pass FPOF ($\text{RESPAT}_{\text{no}}$) with the sample size k for retrieving the top-K outliers in all the datasets having a fixed size transaction except `connect`. Interestingly, we observe that our approach approximatively retrieves in a data stream the outliers that would be obtained by storing all the data observations (using 2-STEP). As expected, the accuracy of the two approaches increases rapidly with the sample size k. The gain of RESPAT is very strong between 10 and 1,000 but,

much lower between 1000 and 1,000,000. The higher the number of outliers K, the more accurate the approach. On the one hand, the imprecision of the sampling only has an impact around the Kth outlier. On the other hand, a high K makes it possible to build a more representative sample from the first K transactions (which are all considered as outliers at the beginning of the pass). The latter explains why our approach is slightly less stable and less accurate than a non-streaming context with 2-STEP.

6 Conclusion

This paper presents the first frequent pattern sampling approach in data streams based on reservoir sampling. The strength of our generic algorithm is do deal with any damping function while having a space complexity only linear with the sample size. We have also shown how to optimize this algorithm for three damping functions usuallly considered in the state-of-the-art. Surprisingly, our theoretical analysis proves that these algorithms work best when the damping is low. Of course, they turn out to be slower than the two-step random procedure, but they require a limited memory space essential to process data streams or to process datasets that do not fit in memory. Finally, a use case illustrates the practical interest of an online pattern sample to detect outliers in one pass. Of course, this simple outlier detection method could be improved by keeping more transactions as candidate outliers and by using a bound to get statistical guarantees on rejected transactions as done in [14]. We would like to extend our approach to other languages and other interestingness measures. In both cases, the challenge lies in extending the index operator for mapping each value to a specific occurrence within a data observation. Finally, it would be interesting to consider a dynamic damping function for learning with drift detection [13].

References

1. Aggarwal, C.C.: On biased reservoir sampling in the presence of stream evolution. In: Proc. of VLDB. pp. 607–618. VLDB Endowment (2006)
2. Aggarwal, C.C.: Managing and mining sensor data. Springer Science & Business Media (2013)
3. Al Hasan, M., Zaki, M.J.: Output space sampling for graph patterns. Proc. of VLDB **2**(1), 730–741 (2009)
4. Babcock, B., Datar, M., Motwani, R.: Sampling from a moving window over streaming data. In: Proc. of ACM-SIAM symposium on Discrete algorithms. pp. 633–634. Society for Industrial and Applied Mathematics (2002)
5. Boley, M., Lucchese, C., Paurat, D., Gärtner, T.: Direct local pattern sampling by efficient two-step random procedures. In: Proc. of KDD. pp. 582–590. ACM (2011)
6. Calders, T., Dexters, N., Gillis, J.J., Goethals, B.: Mining frequent itemsets in a stream. Information Systems **39**, 233–255 (2014)
7. Chi, Y., Wang, H., Yu, P.S., Muntz, R.R.: Moment: Maintaining closed frequent itemsets over a stream sliding window. In: Proc. of ICDM. pp. 59–66. IEEE (2004)

8. De Francisci Morales, G., Bifet, A., Khan, L., Gama, J., Fan, W.: IoT Big Data stream mining. In: Proc. of KDD. pp. 2119–2120 (2016)
9. Diop, L., Diop, C.T., Giacometti, A., Li, D., Soulet, A.: Sequential pattern sampling with norm-based utility. Knowledge and Information Systems **62**(5), 2029–2065 (2020)
10. Efraimidis, P.S., Spirakis, P.G.: Weighted random sampling with a reservoir. Information Processing Letters **97**(5), 181–185 (2006)
11. Gaber, M.M., Zaslavsky, A., Krishnaswamy, S.: Mining data streams: a review. ACM Sigmod Record **34**(2), 18–26 (2005)
12. Gama, J.: A survey on learning from data streams: current and future trends. Progress in Artificial Intelligence **1**(1), 45–55 (2012)
13. Gama, J., Medas, P., Castillo, G., Rodrigues, P.: Learning with drift detection. In: Brazilian symposium on artificial intelligence. pp. 286–295. Springer (2004)
14. Giacometti, A., Soulet, A.: Frequent pattern outlier detection without exhaustive mining. In: Proc. of PAKDD. pp. 196–207. Springer (2016)
15. Giannella, C., Han, J., Pei, J., Yan, X., Yu, P.S.: Mining frequent patterns in data streams at multiple time granularities. Next generation data mining **212**, 191–212 (2003)
16. Jiang, N., Gruenwald, L.: Research issues in data stream association rule mining. ACM Sigmod Record **35**(1), 14–19 (2006)
17. Jin, R., Agrawal, G.: Frequent pattern mining in data streams. In: Data Streams, pp. 61–84. Springer (2007)
18. Karim, M.R., Cochez, M., Beyan, O.D., Ahmed, C.F., Decker, S.: Mining maximal frequent patterns in transactional databases and dynamic data streams: A spark-based approach. Information Sciences **432**, 278–300 (2018)
19. Krempl, G., Žliobaite, I., Brzeziński, D., Hüllermeier, E., Last, M., Lemaire, V., Noack, T., Shaker, A., Sievi, S., Spiliopoulou, M., et al.: Open challenges for data stream mining research. ACM SIGKDD explorations newsletter **16**(1), 1–10 (2014)
20. Li, K.H.: Reservoir-sampling algorithms of time complexity O(n(1+log(N/n))). ACM Transactions on Mathematical Software (TOMS) **20**(4), 481–493 (1994)
21. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proc. of VLDB. pp. 346–357. Elsevier (2002)
22. Martin, T., Francoeur, G., Valtchev, P.: CICLAD: A fast and memory-efficient closed itemset miner for streams. In: Proc. of KDD. pp. 1810–1818 (2020)
23. Raïssi, C., Poncelet, P.: Sampling for sequential pattern mining: From static databases to data streams. In: Proc. of ICDM. pp. 631–636. IEEE (2007)
24. Ramírez-Gallego, S., Krawczyk, B., García, S., Woźniak, M., Herrera, F.: A survey on data preprocessing for data stream mining: Current status and future directions. Neurocomputing **239**, 39–57 (2017)
25. Rehman, M.H.u., Liew, C.S., Wah, T.Y., Khan, M.K.: Towards next-generation heterogeneous mobile data stream mining applications: Opportunities, challenges, and future research directions. Journal of Network and Computer Applications **79**, 1–24 (2017)
26. Tanbeer, S.K., Ahmed, C.F., Jeong, B.S., Lee, Y.K.: Sliding window-based frequent pattern mining over data streams. Information sciences **179**(22), 3843–3865 (2009)
27. Vitter, J.S.: Random sampling with a reservoir. ACM Transactions on Mathematical Software (TOMS) **11**(1), 37–57 (1985)
28. Wong, R.C.W., Fu, A.W.C.: Mining top-k frequent itemsets from data streams. Data Mining and Knowledge Discovery **13**(2), 193–217 (2006)