Knowl Inf Syst (2017) 53:391–422
DOI 10.1007/s10115-017-1045-1

REGULAR PAPER

Time-weighted counting for recently frequent pattern mining in data streams

Yongsub Lim[1] · **U. Kang**[2]

Received: 22 April 2016 / Revised: 20 January 2017 / Accepted: 17 March 2017 /
Published online: 22 March 2017
© Springer-Verlag London 2017

Abstract How can we discover interesting patterns from time-evolving high-speed data streams? How to analyze the data streams quickly and accurately, with little space overhead? How to guarantee the found patterns to be self-consistent? High-speed data stream has been receiving increasing attention due to its wide applications such as sensors, network traffic, social networks, etc. The most fundamental task on the data stream is frequent pattern mining; especially, focusing on recentness is important in real applications. In this paper, we develop two algorithms for discovering recently frequent patterns in data streams. First, we propose TwMinSwap to find top-k recently frequent *items* in data streams, which is a deterministic version of our motivating algorithm TwSample providing theoretical guarantees based on item sampling. TwMinSwap improves TwSample in terms of speed, accuracy, and memory usage. Both require only $O(k)$ memory spaces and do not require any prior knowledge on the stream such as its length and the number of distinct items in the stream. Second, we propose TwMinSwap- Is to find top-k recently frequent *itemsets* in data streams. We especially focus on keeping self-consistency of the discovered itemsets, which is the most important property for reliable results, while using $O(k)$ memory space with the assumption of a constant itemset size. Through extensive experiments, we demonstrate that TwMinSwap outperforms all competitors in terms of accuracy and memory usage, with fast running time. We also show that TwMinSwap- Is is more accurate than the competitor and discovers recently frequent itemsets with reasonably large sizes (at most 5–7) depending on datasets. Thanks to TwMinSwap and TwMinSwap- Is, we report interesting discoveries in real world data streams, including the difference of trends between the winner and the loser of U.S. presidential candidates, and temporal human contact patterns.

✉ U. Kang
ukang@snu.ac.kr

[1] Big Data Tech. Lab, SK Telecom, Seongnam, Republic of Korea

[2] Department of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea

Keywords Data stream · Time-weighted counting · Sampling · Frequent items · Frequent itemsets · Hot items · Top-k items

1 Introduction

How can we discover currently emerging patterns in high-speed data streams, like keyword streams from social networks or click streams from e-commerce sites? How to track them in real time with high accuracy and small memory requirements? How can we guarantee results to be self-consistent? These questions are directly related to recently frequent pattern mining in a data stream.

Formally, a data stream is defined by a sequence of transactions, each of which is a set of items, arriving one by one. Usually, the length of the stream and the number of distinct items are very large numbers, possibly infinite. Due to this massiveness, it is impossible to store all the information from the stream, and thus it becomes important to efficiently use memory spaces. As a result, most data stream mining algorithms [29,30] perform approximation rather than exact computation, and the followings are generally required [6]. First, the stream should be scanned as few times as possible: only one scan is enough for many recent algorithms. Second, the amount of used memory spaces should be limited and independent of the number of distinct items and the stream length, e.g. $O(k)$ space complexity for finding top-k frequent items. Third, processing a transaction at each time should be fast because the rate of transaction arrival can be bursty, e.g. Internet traffic may be exploded by network anomalies like DDoS (Distributed Denial of Service) attacks. Fourth, an up-to-date result should be available on demand. These requirements allow that data stream mining algorithms run in real time with small memory spaces.

A number of studies [9,10,17,21,32,38] have shown efficacy of their methods in finding frequent patterns including items and itemsets in a data stream, but still it is not clear whether they can find *recent* frequent patterns (items or itemsets) correctly. Although a pattern whose frequency decreases over time tends to have a small count by construction of algorithms, it is not done explicitly. Another problem is that among discovered frequent patterns, it is hard to know when they have become frequent. Depending on applications, the problem is crucial. For example, when we monitor keywords mentioned in SNS, it is important to know which one is the current trend and which one is the past trend. Also, in an e-commerce site, a manager would be interested in sets of products co-purchased frequently not just in the all days but in recent days to understand consumers' current needs correctly. To overcome this weakness, finding recent frequent patterns from a data stream has been also studied [1,13,18]. However, they have limitations in accuracy, running time, and memory usage.

In this paper, we propose two recently frequent pattern mining algorithms: TwMinSwap for items and TwMinSwap- Is for itemsets. The idea is to count items or itemsets with time-weighting, which means that a value of an item or itemset decreases over time.

TwMinSwap is a deterministic version of our motivating algorithm TwSample which is a sampling-based randomized algorithm with theoretical guarantees. TwMinSwap improves TwSample in terms of speed, accuracy, and memory usage. Both algorithms only require $O(k)$ space complexity. Especially, TwMinSwap requires no other parameter than k and α (time-decaying factor), and this simplicity enables to not only save memory spaces but also reduce per-item processing time. Table 1 compares our proposed TwMinSwap with other competitors, and Fig. 1 shows the plots of memory usage versus error in estimated time-weighted counts for them. TwMinSwap outperforms the others in terms of precision and recall, time-weighted count estimation, and memory usage; its speed is com-

Table 1 Comparison of performance of TwMinSwap, TwSample and competitors

	[Recommended]		Competitors	
	TwMinSwap[a]	TwSample[b]	TwFreq[c]	TwHCount[d]
Precision and Recall	**Highest**	Low	~~Lowest~~	High
Error in TwCount	**Lowest**	Medium	~~Highest~~	**Lowest**
Memory Usage	**Smallest**	~~Largest~~	Small	~~Largest~~
Time	Fast	~~Slowest~~	Medium	**Fastest**

For each row, we write the best in bold and the worst with the canceled line. Our TwMinSwap outperforms the others in precision and recall, error in estimation of time-weighted counts, and memory usage, where in running time, it is the second best one. The precision and recall and the error are defined in Eqs. 2 and 3, respectively. For more related works, we refer to [13,18,31]

[a] Section 3.2
[b] Section 3.1
[c] Zhang et al. [39]
[d] Chen and Mei [9]

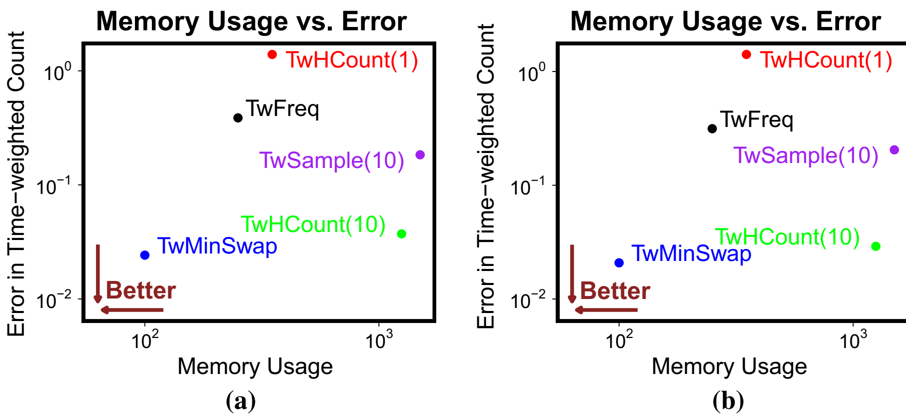


Fig. 1 Our proposed algorithm TwMinSwap outperforms the others in both memory usage and error of estimated time-weighted counts of top-k items. Despite comparable estimation of TwHCount(10) to that of TwMinSwap, TwHCount(10) requires much more memory space. **a** Static distribution, **b** dynamic distribution

parable to the fastest competitor TwHCount requiring large memory spaces (see also Fig. 5).

Our second algorithm TwMinSwap- Is is devised for finding the top-k recently frequent itemsets in data streams. TwMinSwap- Is always keeps at most k itemsets, and outputs self-consistent results, i.e. the Apriori property[1] holds in the found top-k itemsets, which makes the results more reliable. Also the maximum size of itemsets found remains reasonable, while the competitor's can be arbitrary large. Table 2 briefly compares TwMinSwap- Is and the competitor.

Conducting extensive experiments, we demonstrate that TwMinSwap finds top-k time-weighted frequent items with small memory spaces, and small error in terms of precision and recall, and estimated time-weighted counts. Also applying TwMinSwap to real-world data streams, we show that tracking recently frequent activities and keywords enables to

[1] If an itemset μ is frequent, its every subset $\nu \subseteq \mu$ is also frequent.

Table 2 Comparison of performance of TwMinSwap- Is and the competitor

	TwMinSwap- Is (Sect. 4)	Skip LC- SS [38]
Consistency in Result	Apriori property	No guarantee
Max. Size of Itemsets	Reasonable (5–7)	Arbitrary (10–20)

TwMinSwap- Is guarantees the Apriority property for its result, which is not the case for Skip LC- SS, and outputs itemsets with the maximum size 5–7. This size of the output itemsets is reasonable because a large portion of transactions in the used data have a size smaller than 7. Note that by construction Skip LC- SS can result in itemsets of an arbitrarily large size

discover sudden bursts of attentions to currently hot events and trends in real time. We also evaluate TwMinSwap- Is in accuracy and non-triviality of results. TwMinSwap- Is outputs top-k itemsets whose time-weighted counts are highly correlated with the ground truth, and the result includes an itemset of a size at most 5–7 where a large proportion of transactions in data are of a size less than 7: depending on datasets, up to 95% of transactions have a size smaller than or equal to 7.

Our contributions are summarized as follows.

1. *Method* Based on time-weighted counting, we propose TwMinSwap and TwMinSwap-Is for finding top-k recently frequent items and itemsets, respectively. TwMinSwap with $O(k)$ memory requirement is a deterministic variation of our motivating randomized algorithm TwSample. Moreover, extending TwMinSwap to itemsets, we develop TwMinSwap- Is which guarantees the Apriori property in the results.
2. *Performance* We show that TwMinSwap outperforms in precision, accuracy and time-weighted count estimation, and that the speed of TwMinSwap is comparable to that of the fastest competitor TwHCount. Especially, TwMinSwap is $1.5\times$ better in time-weighted count estimation and $2.5\times$ better in memory usage than the second best competitors. We also show that TwMinSwap- Is finds itemsets whose time-weighted counts show high correlation with the true values, and whose sizes are non-trivial, i.e. 5–7.
3. *Discovery* We apply TwMinSwap and TwMinSwap- Is to real-world data streams and show interesting discoveries. They include difference of trends between the winner and the loser of U.S. presidential candidates, and different human contact patterns in real life.

The rest of the paper is organized as follows. In Sect. 2, we give related works including algorithms with and without time-weighting. In Sect. 3, we describe and analyze TwSample and TwMinSwap for recently frequent items, and in Sect. 4 we develop TwMinSwap- Is for recently frequent itemsets. In Sects. 5 and 6, performance evaluation results, including comparison with competitors, are presented for TwMinSwap and TwMinSwap- Is, respectively. In Sect. 7, we show the discovery results of applying TwMinSwap to real-world data streams. Finally, we conclude our work in Sect. 8.

Table 3 lists the symbols frequently used in this paper.

2 Related works

2.1 Finding frequent items

There have been numerous studies to find frequent items from a data stream, including not only developing methods but also comparing them [10,32]. Here, we classify them into three categories as follows.

Table 3 Table of symbols

Symbol	Description
N	Stream length
n	# of distinct items
k	# of (time-weighted) frequent items
α	Time-decaying factor ($0 < \alpha < 1$)
$u, v; \mu, \nu$	Item; itemset
2^{μ}	Power set of μ
c, c_u	Counter (of u)
T_u	Set of timestamps that u occurs
$W(u)$	Time-weighted count of u
$L(u)$	Penalized time-weighted count of u
$t, t_i/t_{cur}$	Timestamps/current timestamp
K	Set of discovered (time-weighted) frequent items
λ	Penalty term for $L(u)$
σ	Increase ratio of λ

Sampling-based Approach This approach involves a probabilistic process, and obtained items are random samples over all the items occurring in a data stream. Vitter [37] proposed a uniform sampling method from a data stream by which, in essence, higher frequency items are sampled much more than lower ones. Improving the space requirement for the sampling, Gibbons and Matias [17] invented a method called concise sampling which efficiently represents sampled items. By slightly modifying the method, they also proposed counting sampling to estimate the frequency of each item more accurately. A similar approach was adopted in [33] to obtain high frequency items.

Counter-based Approach A very basic form of the counter based approach is MAJORITY that finds the majority item in a stream if it exists [3,16]. By generalizing MAJORITY, Misra and Gries [36] developed a method to find items occurring at least N/k times and it was improved in per-item processing time by Demaine et al. [14] and Karp et al. [22]. LOSSYCOUNTING [33] does the same job, but it additionally guarantees that no item whose count is less than $N(1/k - \epsilon)$ is reported. SPACESAVING [35] reduces the space requirement not only for an arbitrary data distribution but also for a Zipf distribution.

Sketch-based Approach The sketch-based approach is usually based on using multiple hash functions to map incoming items to a hash table. This can be also understood as maintaining a list of independent counters where each counter is shared by a few items, and the sharing is determined by the hash functions. Charikar et al. [8] proposed COUNTSKETCH that computes items appearing at least $N/(k+1)$ times with probability $1 - \delta$ while requiring $O(k/\epsilon^2 \log N/\delta)$ memory spaces. The space requirement was improved by COUNTMIN [11]. GROUPTEST [12] was developed for a hot item query, which groups items and assumes one frequent item in each group. Jin et al. [20] improved GROUPTEST in space and their algorithm guarantees the minimum count of items outputted.

2.2 Finding recent frequent items

Despite many algorithms to find frequent items from a data stream, researchers have agreed that recent items are more important than old ones and answering a query of finding recently frequent items is often required. Below, we introduce two approaches for the purpose.

Sliding Window-based Approach This approach divides recent times into window blocks, performs counting for items in the windows, and aggregates them. Golab et al. [18] proposed a method based on basic window blocks for identifying frequent items in packet streams in which item frequency is known to follow a power-law distribution. They also extended the work to the more general case that the item frequency follows a multinomial distribution [19]. There have been works using windows of various sizes. Arasu and Manku [1] provided deterministic and randomized algorithms for ϵ-approximate quantiles over sliding windows. Dallachiesa and Palpanas [13] studied the problem in ad hoc time windows. They use overlapping window blocks whose sizes exponentially grow by which a query for frequent items during a certain recent time period can be answered more accurately.

Time-aware-based Approach This approach implicitly considers the recentness. Liu et al. [31] proposed a pruning method, a key operation of counter-based algorithms, considering time information so that an older item is more likely to be pruned than a more recent one when the memory becomes full. A similar approach has been examined in [9,39,40]. All of them adopt a time fading factor in developing methods to find frequent items which decreases a weight of an item over time. As a result, recent items have more weights, leading to more accurate results. Our proposed algorithms, whose preliminary version appeared in [28], in this paper also belong to this category. We show that time-weighted counting can be done via sampling which guarantees its accuracy in expectation, and propose our main algorithm TwMinSwap via derandomization.

2.3 Finding frequent itemsets

In this problem, an object from a data stream is a set of items called a *transaction*, and every subset of the transaction is a candidate to be found as a frequent itemset. This means that there is an exponentially many candidates on the length of a transaction, and thus the problem becomes more challenging than frequent item mining.

LossyCounting has been refined for frequent itemset mining and that with an information decaying parameter [7,33]. The problem has been studied with various aspects. Examples include focusing maximal frequent itemsets [23,27], targeting streams with bursty transactions [38], and window-based approaches [5,26]. Several studies [6,7] have taken a similar approach of our time-weighted counting. They preserve the Apriori property using a prefix-tree lattice structure, but its space requirement increases as the number of distinct items gets larger. In contrast, our proposed algorithm TwMinSwap- Is maintains only $O(k)$ entries for the top-k recently frequent itemsets as well as keeping the Apriori property.

3 Recently frequent item mining

In this section, we propose our method for finding time-weighted top-k items from a data stream. The main idea is derandomization of a sampling-based algorithm. As a result, we propose a deterministic algorithm TwMinSwap, which requires only $O(k)$ memory spaces where k is the number of time-weighted frequent items that we want to find. To develop TwMinSwap, we first propose a sampling-based randomized algorithm TwSampleto guarantee performance in expectation, which helps understand a theoretical background of TwMinSwap. However, the probabilistic nature of TwSample requires several independent sampling sessions to achieve high accuracy, leading to large memory spaces and slow running time. On the other hand, TwMinSwap achieves high accuracy with fast running time while using only a single session.

We start with the definition a time-weighted count of an item [9,39,40].

Definition 1 (*Time-weighted Count*) Let u be an item occurring in a data stream at times $t_1, \ldots, t_c$. The time-weighted count of the item u is defined by

$$W(u) = \sum_{i=1}^{c} \alpha^{t_{cur}-t_i},$$

where $0 < \alpha < 1$ is a decaying parameter and t_{cur} is the current time.

3.1 Randomized algorithm

To develop a sampling-based randomized algorithm, we first define a penalized time-weighted count as follows.

Definition 2 (*Penalized Time-weighted Count*) Let u be an item occurring in a data stream and let T_u be a set of the times at which u has occurred. The penalized time-weighted count of the item u is defined by

$$L(u) = \sum_{t \in T_u} \alpha^{t_{cur}-t+\lambda-1},$$

where $0 < \alpha < 1$ is a decaying parameter, t_{cur} is the current time, and $\lambda \geq 1$ is a default penalty term for items just arriving.

Let the sequence of items till time t_{cur} be $u_1, \ldots, u_{t_{cur}}$. Given $0 < \alpha < 1$ and $\lambda \geq 1$, our randomized algorithm samples each item u_t with probability $\alpha^{t_{cur}-t+\lambda-1}$. This is incrementally done with increasing λ over time to ensure that the number of distinct items in the samples is at most k. Indeed, our algorithm can be understood as an extension of the uniform sampling in a data stream [17] to a time-weighted sampling. Below, we call that u is monitored if u has been sampled at least once.

Precisely, our randomized algorithm TwSample requires three parameters: the maximum number k of monitored items, the time-weighting factor α, and the increase ratio σ for the penalty term λ. Also there are three pieces of information incrementally updated: the penalty term λ, the set K of monitored items, and counters c_v associated with each $v \in K$. Initially, $\lambda = 1$ and $K = \emptyset$. A new item u currently arriving is determined whether sampled or not with probability $\alpha^{\lambda-1}$. The sampling is done as follows: if u is currently monitored, i.e. $u \in K$, c_u is incremented by 1; otherwise u is added to K with its associated counter $c_u = 1$. After the sampling, if $|K| > k$, *downsampling* is applied to all the sampled items so far with increasing λ. Precisely, λ is incremented by σ, and for each $v \in K$, c_v is updated by a random number drawn from $binomial(c_v, \alpha^\sigma)$.[2] If c_v becomes 0, v is evicted from K. This downsampling is repeated until $|K| \leq k$. Lastly, whenever one time step passes, all items in K are unconditionally downsampled as follows: for each $v \in K$, $c_v = binomial(c_v, \alpha)$. Algorithm 1 fully describes TwSample.

Effect of σ Essentially, σ determines the sampling rate for evicting existing items so that the number of sampled items does not exceed k. As σ gets smaller, the amount of decrements of each item count in Line 14 is reduced, leading to longer time for the eviction process in Line 12 to 15. Accuracy may increase since we do not evict more than required. On the

[2] Here, $binomial(\omega, \theta)$ denotes a binomial random variable with the number ω of independent trials and the success probability θ.

Algorithm 1: TWSAMPLE: Randomized Time-Weighted Counting

```
Input: A data stream S, a number k of counters, a decaying parameter α, and increase ratio σ of the
       penalty term.
Output: Top-k time-weighted frequent items K (continuously updated)
1  λ ← 1.
2  K ← ∅.
3  foreach new item u from S do
4  |   Downsampling(α).
5  |   if bernoulli(α^(λ−1)) = 1 then
6  |   |   if u ∈ K then
7  |   |   |   c_u ← c_u + 1.
8  |   |   else
9  |   |   |   K ← K ∪ {u}.
10 |   |   |   c_u ← 1.
11 |   |   end
12 |   |   while |K| > k do
13 |   |   |   λ ← λ + σ.
14 |   |   |   Downsampling(α^σ).
15 |   |   end
16 |   end
17 end
18 Subroutine Downsampling (θ)
19 foreach v ∈ K with counter c_v do
20 |   c_v ← binomial(c_v, θ).
21 |   if c_v = 0 then
22 |   |   K ← K \ {v}.
23 |   end
24 end
```

other hand, as σ gets larger, the eviction process finishes quickly, although the accuracy may decrease since we may evict more than one item.

The following lemma shows that the sampling probability of any individual item is equal to its penalized time-weight.

Lemma 1 *At time t_{cur} with the penalty term λ, each item u occurring at time $t \le t_{cur}$ has been sampled with probability*

$$\Pr\left[u \text{ is sampled}\right] = \alpha^{t_{cur}-t+\lambda-1}.$$

Proof Let u be a new item at $t = 1$ and $t_{cur} = 1$. It is unconditionally sampled because $\lambda = 1$, and all counters are empty. In other words, u is sampled with the probability $\alpha^{t_{cur}-t+\lambda-1} = 1$.

Let $t_{cur} \ge 1$. Assume that the lemma holds for time t_{cur}. That is, for each sample v at time $t \le t_{cur}$, it has been sampled with probability $\alpha^{t_{cur}-t+\lambda-1}$. Let us consider the process for $t_{next} = t_{cur} + 1$. In Line 4, all samples are downsampled with probability α. This means that after Line 4, remaining samples are with probability $\alpha^{t_{cur}+1-t+\lambda-1}$ which matches the sampling probability at time $t_{next} = t_{cur} + 1$.

Next, we verify from Line 5 to 11. Let u be a new item occurring at $t = t_{next}$. Clearly, u is sampled with probability $\alpha^{t_{next}-t+\lambda-1} = \alpha^{\lambda-1}$ by Line 5, regardless of whether u is currently monitored or not. Let us verify the downsampling. Let d be the number of iterations by the while statement in Line 12. Then, after the downsampling, each sample experiences re-sampling with probability $\alpha^{d\sigma}$, and thus each sample is with probability $\alpha^{t_{next}-t+\lambda+d\sigma-1}$ which matches the update of $\lambda = \lambda + d\sigma$.

Note that the sampling probability does not increase over time since λ increases or remains the same at every time step. Hence, an item that is not a sample at a certain time cannot be a sample in the future. □

From Lemma 1, we obtain the following corollary which states that the expected count of a distinct item is equal to its penalized time-weighted count.

Corollary 1 *At any time t_{cur} in* TWSAMPLE*, the expectation of a counter c_u of a monitored item $u \in K$ is*

$$\mathbb{E}[c_u] = L(u) = \sum_{t \in T_u} \alpha^{t_{cur}-t+\lambda-1},$$

where T_u is a set of times at which u has occurred.

Since we know λ at any time, we can compute the expected time-weighted count for a monitored item u. That is, the estimated time-weighted count of $u \in K$ becomes $\alpha^{1-\lambda} c_u$. Next, we show that as an item becomes more insignificant, the probability that it is not monitored increases exponentially.

Lemma 2 *At any time, the probability p_u that an item u is monitored satisfies the following inequality:*

$$p_u \geq 1 - \exp(-L(u)/2).$$

Proof Note that $p_u = \Pr[c_u > 0]$ since $c_u = \sum_{i=1}^{|T_u|} c_u(i)$ is a random variable where $c_u(i)$ indicates whether the i-th occurrence of u is sampled or not. Applying the Chernoff bound, we obtain the following inequality:

$$p_u = \Pr[c_u > 0] = 1 - \Pr[c_u = 0] \geq 1 - \exp(-\mathbb{E}[c_u]/2).$$

Since $\mathbb{E}[c_u] = L(u)$ by Corollary 1, the proof is done. □

Although TWSAMPLE is simple and provides the theoretical guarantees of its output, its performance may be degraded due to its probabilistic nature. First, the running time may become slow because the number of iterations for the downsampling with increasing λ is not fixed and the time for drawing random variables from $binomial(c, \theta)$ used in the downsampling depends on c. Second, discovered top-k items and the associated counters may be inaccurate due to unintendedly large λ. This inaccuracy can be resolved by maintaining s number of independent sessions each of which monitors at most k items, but it leads to more memory spaces and longer running time. In the next section, we propose a deterministic variation of TWSAMPLE, which is fast and requires a single session of monitored items of size at most k.

3.2 Deterministic algorithm

In this section, we propose TWMINSWAP for efficient top-k time-weighted frequent items discovery. This algorithm is a deterministic version of TWSAMPLE with truncating insignificantly old items. Concretely, the time-decaying factor α has the same meaning as that in TWSAMPLE, but affects counts of items in a deterministic manner.

The main idea is to record the expected number of samples for each item directly instead of applying the random process. Concretely, for an item u occurring at $t \leq t_{cur}$, instead of incrementing c_u by 1 with probability $\alpha^{t_{cur}-t}$, we increment c_u by $\alpha^{t_{cur}-t}$ with probability 1.

Algorithm 2: TwMinSwap: Deterministic Time-Weighted Counting

Input: A data stream S, the number of counters k, and a decaying parameter α.
Output: Top-k time-weighted frequent items K (continuously updated)

```
1  K ← ∅.
2  t_cur ← 0.
3  foreach new item u from S do
4  |  t_cur ← t_cur + 1.
5  |  foreach v ∈ K do
6  |  |  c_v ← c_v × α.
7  |  end
8  |  if u ∈ K then
9  |  |  c_u ← c_u + 1.
10 |  else if |K| < k then
11 |  |  K ← K ∪ {u}.
12 |  |  c_u ← 1.
13 |  else
14 |  |  v* ← argmin_{v∈K} c_v.
15 |  |  if c_{v*} < 1 then
16 |  |  |  K ← K \ {v*} ∪ {u}.
17 |  |  |  c_u ← 1.
18 |  |  end
19 |  end
20 end
```

Then, the downsampling with rate θ becomes that for each monitored item $v \in K$, $c_v = \theta c_v$. With this deterministic scenario, however, we encounter a problem when all counters become full. Precisely, because the expected count for an item occurring at least one time in the stream never becomes 0, it needs pruning for dropping an insignificant monitored item to start monitoring a new item. We propose a simple heuristic for the pruning which does not require additional memory spaces, leading to smaller memory usage compared with the previous approaches [9,39,40]. We note that this proposed pruning method here corresponds to decreasing the sampling rate by increasing λ in TwSample.

Details of the heuristic for the pruning are as follows. Let K be a set of currently monitored items where $|K| = k$, and u be an item just arriving from a data stream. Our pruning method first finds an item $v^* \in K$ having the minimum time-weighted count $c_{v^*} = \min_{v \in K} c_v$. If $c_{v^*} < 1$, we drop v^* and start monitoring u with initial count 1; otherwise, u is ignored. This swapping of u and v^* makes sense because c_{v^*} is computed by a few most recent occurrences of v^* but smaller than the effect 1 by the single occurrence of u at t_{cur}. In this strategy, a newly added item is not evicted for the next r timesteps even though it never occurs where r is the number of items with a count less than 1 at its addition time. The overall procedure of TwMinSwap is described in Algorithm 2.

Advantages of TwMinSwap are summarized as follows. First, its memory usage is small. For each item, only an item identifier and its time-weighted count are maintained. This simple structure enables to reduce per-item computation compared with similar counter-based approaches [39], and greatly saves memory spaces compared with sketch-based algorithms [9]. Second, TwMinSwap requires the minimal parameters: k and α. This especially gives the benefit of reducing efforts for parameter tuning in practice. Third, in contrast to TwSample, TwMinSwap guarantees to output k number of items so long as $N \geq k$.

Per-item Processing Time The main time-consuming operations are: (1) computing an item with the minimum count, and (2) multiplying α to all counters each of which requires scanning

K. The second operation can be eliminated by maintaining the most recent time for each item when it occurs [39]. In this way, we benefit in speed when a new item is already monitored. In contrast, the first operation taking $O(k)$ time is unavoidable. Although an efficient data structure was proposed for the case without time-weighting [14], it cannot be applied to our time-weighted case since our counters record real numbers with which difference between two numbers is unfixed. As a result, TwMinSwap requires $O(k)$ computation for each iteration.

3.2.1 Analysis

We analyze TwMinSwap especially for the condition that a monitored item is not evicted from K. More precisely, Lemmas 3 and 4 state that TwMinSwap will not evict items whose frequencies of occurrences are above certain thresholds for a general and a power-law item distribution cases, respectively.

Lemma 3 *Let* $0 < \alpha < 1$ *be a time-decaying parameter of* TwMinSwap*. Any item with count* $c \geq 1$ *will not be evicted from* K *if it occurs at least once per every* $1 - \log_\alpha \gamma$ *times where* $\gamma = \min\{1+\alpha, c\}$.

Proof Assume that $v \in K$ with count $c_v = c \geq 1$ at a certain time occurs for every d times. Note that any item whose count is at least 1 is never evicted from K by construction. Let us define the following function.

$$g(r+1) = (g(r)\alpha + 1)\,\alpha^{d-1},$$

where $g(1) = c\alpha^{d-1}$. It is clear that $g(r)$ is c_v after $rd - 1$ time steps. Since c_v always decreases from $(r-1)d + 1$ to $rd - 1$, it suffices to show that $g(r) \geq 1$ with $d \leq 1 - \log_\alpha \gamma$ for every $r \geq 1$ where $\gamma = \min\{1+\alpha, c\}$.

For $r = 1$, assume that $c \leq 1 + \alpha$; then,

$$g(1) = c\alpha^{d-1} \geq c\alpha^{-\log_\alpha c} = 1.$$

Assume that $c > 1 + \alpha$.

$$g(1) = c\alpha^{d-1} > (1+\alpha)\alpha^{d-1} \geq (1+\alpha)\alpha^{-\log_\alpha(1+\alpha)} = 1.$$

For $r > 1$, assume that $g(r-1) \geq 1$; then,

$$\begin{aligned} g(r) &= (g(r-1)\alpha + 1)\,\alpha^{d-1} \geq (\alpha+1)\alpha^{d-1} \\ &\geq \min\{1+\alpha, c\} \times \alpha^{-\log_\alpha \min\{1+\alpha, c\}} = 1. \end{aligned}$$

□

Below, we show which item is expected not to be evicted from K before the next occurrence of the item for a power-law item distribution.

Lemma 4 *Let n be the number of items and let us consider a power-law item distribution with the exponent* 2*. Any monitored item* $i \leq \sqrt{(1 - \log_\alpha(1+\alpha))/1.7}$ *with count* $c_i \geq \alpha^{1-1.7i^2}$ *is expected not to be evicted.*

Proof For an item $i \in [1, n]$, the probability that it occurs in a stream is

$$\Pr[\text{i occurs}] = i^{-2}/Z,$$

where Z is the normalization constant. Hence, the expected number of occurrences before i occurs becomes $i^2 Z$. Since an occurrence of i obeys $geometric(i^{-2}/Z)$, the probability that i does not occur during $i^2 Z$ time steps is less than $1/e$.

Assume that $c_i \leq \alpha + 1$. The following guarantees in expectation that i will not be evicted by Lemma 3:

$$1 - \log_\alpha c_i \geq i^2 Z \iff 1 - i^2 Z \geq \log_\alpha c_i \iff \alpha^{1-i^2 Z} \leq c_i.$$

The feasible i should satisfy

$$\alpha^{1-i^2 Z} \leq \alpha + 1 \iff i \leq \sqrt{\frac{1 - \log_\alpha(1+\alpha)}{Z}}.$$

Assume $c_i > \alpha + 1$. The following inequality guarantees in expectation that i will not be evicted by Lemma 3:

$$1 - \log_\alpha(1+\alpha) \geq i^2 Z \iff i \leq \sqrt{\frac{1 - \log_\alpha(1+\alpha)}{Z}}.$$

Finally, $Z \leq \sum_{j=1}^{\infty} j^{-2} = \pi^2/6 \leq 1.7$, which completes the proof. □

4 Recently frequent itemset mining

In this section, we describe TwMinSwap- Is to find the top-k time-weighted frequent itemsets in data streams. The overall procedure is similar to TwMinSwap, i.e. maintaining at most k recently frequent itemsets, accepting a new itemset which is likely to be recently frequent, and evicting a maintained itemset whose value is assessed to be smaller than that of the new one.

Algorithm 3 shows the outline of TwMinSwap- Is where $Faded$, $Ordering$, $MostRel$ and $MostCold$ will be defined later. At a high level, $Faded$ calculates $\Theta \subseteq K$ that contains insignificant itemsets; $Ordering$ sorts 2^I in *relevance* to K where 2^I is the power set of I; $MostRel(\Omega)$ returns and removes the most relevant itemset to K in Ω; $MostCold(\Theta)$ returns and removes the most insignificant itemset in Θ.

To define those functions, our approach is to introduce several properties that should be satisfied by the algorithm and propose appropriate implementations of *Ordering, Faded, MostRel* and *MostCold*. We start with defining the following concept.

Definition 3 (*Closeness*) Let K be a set of itemsets. The set K is closed[3] if and only if the Apriori property holds in K, that is, for every $\mu \in K$,

$$\nu \in 2^\mu \setminus \emptyset \implies \nu \in K.$$

Example of Closeness. Let us consider two sets of itemsets as follows: $K_1 = \{a, b, c, d, ab, bc\}$ and $K_2 = \{a, b, c, d, ab, bc, abc\}$. Note that K_1 is closed by definition, but K_2 is not closed because $ac \in 2^{abc}$ is not an element of K_2.

Let K be the set of itemsets maintained by the algorithm; the desired properties of our itemset mining are as follows.

- **(P1)** K should be always closed.
- **(P2)** For $\nu, \mu \in K$, $\nu \subset \mu$ implies $c_\nu \geq c_\mu$ where c_ν and c_μ are the counts of itemsets ν and μ, respectively. Note that $\subset$ denotes a proper subset.

[3] This is a different concept from *closed frequent itemsets* [2].

Algorithm 3: TwMinSwap- Is: Deterministic Time-Weighted Counting for Itemsets.

```
   Input: A data stream S, the number of counters k, and a decaying parameter α.
   Output: Top-k time-weighted frequent itemsets K (continuously updated)
 1 K is initialized by k empty sets with counts c_i...c_k set to 0.
 2 foreach transaction I from S do
 3     foreach μ ∈ K do c_μ ← c_μ × α.
 4     Θ ← Faded(K, I).
 5     Ω ← Ordering(I, K).
 6     while |Ω| > 0 do
 7         ω ← MostRel(Ω).
 8         if ω ∈ K then
 9             c_ω ← c_ω + 1.
10         else if |Θ| > 0 then
11             θ ← MostCold(Θ).
12             K ← K \ {θ} ∪ {ω}.
13             c_ω ← 1.
14         else
15             break
16         end
17     end
18 end
```

The first property **(P1)** is to keep K satisfying the Apriori property, the most fundamental property in frequent itemset mining, in which all subsets of any frequent itemset should be frequent. The second property **(P2)** is a stronger condition than **(P1)**. Note that in the ideal case, the number of occurrences of an itemset cannot be larger than that of its any subset.

To achieve the properties **(P1)** and **(P2)**, we propose two scoring functions for selecting new itemsets from I and evicting faded itemsets in K, respectively. The first one measures irrelevance of an itemset with respect to a set of itemsets, which is formally defined as follows.

Definition 4 (*Irrelevancy of an Itemset*) Given an itemset μ, its irrelevancy with respect to a set K of itemsets is defined as

$$f_K(\mu) = \left|2^{\mu} \setminus K\right| + \frac{1}{|\mu|}.$$

The first term of $f_K(\mu)$ measures the irrelevance of μ to K: the score gets smaller as more subsets of μ are in K. The second term is used for preferring a larger itemset to a shorter one at the same relevance degree. Note that an itemset gets more relevant to K as its irrelevancy is closer to 0. Next, we define the hotness of an itemset in K, which is used to rank itemsets in K.

Definition 5 (*Hotness of an Itemset*) Given an itemset $\mu \in K$ and a new transaction I, the hotness $h(\mu)$ of μ is defined by

$$h(\mu) = \rho_{\mu} + \frac{1}{|\mu|},$$

where $\rho_{\mu} = j$ if μ has the j-th smallest $y_{\mu} = c_{\mu} + \delta(\mu \subseteq I)$ among all itemsets in K, and $\delta(\cdot) = 1$ if $\cdot$ is true and 0 otherwise.

In terms of hotness, an itemset is preferred as its size gets smaller at the same count. This is the same as sorting itemsets in K by their counts first and by their sizes for the ties. Note

that an itemset has a higher hotness score as its time-weighted count becomes larger, i.e. as its recent frequency becomes larger.

With these two scoring functions, we determine the four main operations *Ordering*, *Faded*, *MostRel* and *MostCold* as follows.

- $Ordering$ constructs a priority queue for all itemsets generated by a new transaction I, where an itemset μ's priority is inversely proportional to the irrelevancy $f_K(\mu)$.
- $Faded$ constructs a priority queue for itemsets in K, where an itemset μ's priority is inversely proportional to the hotness $h(\mu)$. During the construction, only itemsets $\mu \in K$ such that $c_\mu < 1$ and $\mu \nsubseteq I$ are considered.
- $MostRel$ and $MostCold$ become pop operations for the priority queues by $Ordering$ and $Faded$, respectively.

In terms of implementations, $Faded$ is easy because given a transaction I, counts and sizes for itemsets considered in $Faded$ do not change from Line 4 to Line 17 of Algorithm 3. As a result, $Faded$ outputs an ordinary queue consisting of itemsets sorted in the ascending order of their hotness. Accordingly, $MostCold$ becomes a pop operation of the queue.

In contrast, there is a performance issue when implementing $Ordering$ and $MostRel$ because the irrelevancy of an itemset changes depending on the previously inserted itemsets (subsets of I) into K. A naive approach is to compute the most relevant itemset per request, that is, whenever $MostRel$ is called. However, the computation is expensive. We present an efficient implementation for $Ordering$ and $MostRel$ in Sect. 4.1.

Lastly, we prove that Algorithm 3 satisfies the properties **(P1)** and **(P2)**. We start with examining addition to and deletion from the itemsets K in Algorithm 3. First, Lemma 5 states that **(P1)** holds if itemsets are added to K by $Ordering$ and $MostRel$ without deleting itemsets from K.

Lemma 5 *Assume that **(P1)** and **(P2)** hold. Let ψ_i be an itemset returned by the i-th call to $MostRel(\Omega)$ where $\Omega = Ordering(I, K)$. Then, $K \cup \{\psi_1, \ldots, \psi_i\}$ is closed for every $0 \le i \le 2^{|I|} - 1$.*

Proof Let $X_i = \{\psi_1, \ldots, \psi_i\}$. For $i = 0$, the statement holds since $X_i = \emptyset$. Assume that for $0 \le i \le 2^{|I|} - 2$, $K'_i = K \cup X_i$ is closed. Letting $\mu = \psi_{i+1}$, suppose that there is $\nu \subset \mu$ such that $\nu \notin K'_{i+1}$. Consider $\nu \subset \chi \subseteq \mu$. It holds $\chi \notin K'_i$; if $\chi \in K'_i$, $\nu \in K'_i$ since K'_i is closed. Let $V = 2^\nu \setminus K'_i$ and $U = 2^\mu \setminus K'_i$. We know $|V| \le |U|$ by $\nu \subset \mu$. We also know $|V| \ne |U|$ because $\chi \notin V$ by $\chi \supset \nu$ and $\chi \in U$ by $\chi \notin K'_i$. As a result, we obtain $|V| < |U|$. Then, ν has a higher priority than μ as follows.

$$f_{K'_i}(\nu) = |V| + \frac{1}{|\nu|} \le |V| + 1 \le |U| < |U| + \frac{1}{|\mu|} = f_{K'_i}(\mu)$$

In other words, μ is not the itemset of the highest priority at the $(i+1)$-th iteration, and consequently $\mu \ne \psi_{i+1}$ which is a contradiction. □

Below, Lemma 6 states that **(P1)** holds if itemsets in K are removed in the order determined by $Faded$ and $MostCold$ without adding itemsets to K.

Lemma 6 *Assume that **(P1)** and **(P2)** hold. Let ϕ_i be an itemset returned by the i-th call to $MostCold(\Theta)$ where $\Theta = Faded(K, I)$. Then, $K \setminus \{\phi_1, \ldots, \phi_i\}$ is closed for every $0 \le i \le |K|$.*

Proof Let $Y_i = \{\phi_1, \ldots, \phi_i\}$. For $i = 0$, the statement holds since $Y_i = \emptyset$. For $K = \emptyset$, the proof is done. Below, we consider $K \ne \emptyset$. Assume that for $0 \le i \le |K| - 1$, $K'_i = K \setminus Y_i$

is closed. Letting $\nu = \phi_{i+1}$, suppose that there is $\mu \supset \nu$ such that $\mu \in K'_{i+1}$. From **(P2)**, $c_\nu \geq c_\mu$; also, if $\mu \subseteq I$, $\nu \subset I$. Then, the following equation holds:

$$h(\nu) = \rho_\nu + \frac{1}{|\nu|} > \rho_\mu + \frac{1}{|\mu|} = h(\mu),$$

where ρ is the rank defined by Definition 5. This implies μ has a higher priority than ν, and thus $\mu \notin K'_{i+1}$ which is a contradiction. □

Lemmas 5 and 6 state that if K is closed, adding itemsets by $\{Ordering, MostRel\}$ to and removing itemsets by $\{Faded, MostCold\}$ from K keep K closed, respectively. Next, we show that **(P1)** is preserved by TwMinSwap- Is where both the addition and the removal operations are performed.

Lemma 7 TwMinSwap- Is *guarantees* ***(P1)***.

Proof Initially, $K = \emptyset$, and thus **(P1)** and **(P2)** hold. Assume that **(P1)** and **(P2)** hold at the beginning of a certain iteration, and let I be a new transaction. Let $X \subseteq 2^I$ be a set of itemsets added to K and $Y \subseteq K$ be a set of itemsets deleted from K during the iteration. We already show that two operations $K = K \cup X$ and $K = K \setminus Y$ preserve **(P1)** by Lemmas 5 and 6, respectively.

Suppose that $K' = K \cup X \setminus Y$ is not closed. This means that there is at least one pair of itemsets $\mu \in K'$ and $\nu \notin K'$ such that $\nu \subset \mu$. There are the following two cases: $\mu \in K \setminus Y$ or $\mu \in X$. Assume that $\mu \in K \setminus Y$; then $\nu \in Y$ since K is closed. But, this implies $K \setminus Y$ is not closed, which is a contradiction to Lemma 6. Assume that $\mu \in X$. It holds $\nu \in Y$ since $\nu \in K \cup X$ by Lemma 5. By the definition of $Faded$, Y does not contain any itemset that is a subset of I; but $\nu \subseteq I$ since $\nu \subset \mu \in X$, which is a contradiction. Consequently, K' is closed. □

The following lemma states that subsets of I already contained in K are considered before those not in K.

Lemma 8 *Assume that* ***(P1)*** *and* ***(P2)*** *hold. Let* ψ_i *be an itemset returned by the* i*-th call to* $MostRel(\Omega)$ *where* $\Omega = Ordering(I, K)$*. If* i *is the largest number such that* $\psi_i \in K$*, then for every* $1 \leq j \leq i$*,* $\psi_j \in K$*.*

Proof The size of any itemset is finite. For $\mu \subseteq I$ contained in K, $|2^\mu \setminus K| = 0$, and for $\nu \subseteq I$ not contained in K, $|2^\nu \setminus K| \geq 1$. Thus, always $f_K(\mu) < f_K(\nu)$. □

With Lemma 8, Algorithm 3 guarantees that all itemsets μ such that $\mu \subseteq I$ and $\mu \in K$ are unconditionally considered, i.e. $c_\mu = c_\mu + 1$ at Line 9 of Algorithm 3. This is because processing such μ does not affect K. In other words, the while loop of Algorithm 3 never reach Line 15 before counts of all itemsets $\mu \in K$ are incremented by 1 at Line 9.

Lemma 9 TwMinSwap- Is *guarantees* ***(P2)***.

Proof Initially, $K = \emptyset$, and thus both **(P1)** and **(P2)** hold. Assume that **(P1)** and **(P2)** hold at the beginning of a certain iteration; let I be a new transaction. Note that $c_\nu = 0$ if $\nu \notin K$. We denote K and c_μ after processing I by K' and c'_μ, respectively. Consider $\mu \in K'$ and its proper subset $\nu \neq \emptyset$.

Case 1 $\mu \notin K$. It means that μ is a new itemset added to K during the iteration, and thus $c'_\mu = 1$ by Line 13. Also it implies $\mu \subseteq I$ and also $\nu \subset I$. Assume $\nu \in K$. We obtain

$c'_\nu = \alpha c_\nu + 1 > 1 = c'_\mu$ from Lemma 8. Assume $\nu \notin K$. We know by Lemma 7 that ν is added to K during the iteration as like μ. That is, $c'_\nu = 1 = c'_\mu$. Consequently, $c'_\mu \leq c'_\nu$.

Case 2 $\mu \in K$. It holds $\nu \in K'$ and $\nu \in K$ due to Lemma 7. In addition, $c_\mu \leq c_\nu$ holds. Assume $\mu \subseteq I$. It also implies $\nu \subseteq I$; then due to Lemma 8, both c_ν and c_μ are guaranteed to increase by 1 in Line 9. Thus, $c'_\mu = \alpha c_\mu + 1 \leq \alpha c_\nu + 1 = c'_\nu$. Assume $\mu \nsubseteq I$; then $c'_\mu = \alpha c_\mu \leq \alpha c_\nu \leq c'_\nu$. □

The following lemma shows that TwMinSwap- Is does not miss itemsets that occur at a rate above a certain threshold.

Lemma 10 *Let* $0 < \alpha < 1$ *be a time-decaying parameter of* TwMinSwap- Is. *Any itemset with count* $c \geq 1$ *will not be evicted from* K *if it occurs at least once per every* $1 - \log_\alpha \gamma$ *times where* $\gamma = \min\{1 + \alpha, c\}$.

Proof Considering an itemset instead of an item, this is proved by the proof of Lemma 3. □

4.1 Implementation for ordering and MostRel

In this section, we describe an efficient implementation for $Ordering$ and $MostRel$.

As stated previously, $Ordering$ returns a priority queue Ω for 2^I where a smaller irrelevancy score implies a higher priority, and $MostRel$ is a pop operation for Ω. The problem is that adding an itemset $\mu \in 2^I$ to K may change irrelevancies of other itemsets. For instance, let $I = \{a, b\}$ and $K = \{\{a\}\}$; the irrelevancy $f_K(\{a, b\})$ of $\{a, b\}$ is currently $3.5 = |2^{\{a,b\}} \setminus K| + 1/|\{a, b\}|$, but if we add $\{b\}$ to K, $f_K(\{a, b\}) = 2.5$.

A naive solution is to search for the itemset of the highest priority whenever $MostRel$ is called; accordingly $Ordering$ performs nothing. This approach, however, results in $O(|2^I||K|^2)$ time complexity for each iteration, which is discussed in Sect. 4.1.3. In contrast, our implementation presented in Sects. 4.1.1 and 4.1.2 reduces it to $O(|K|(|I| + \log|K|))$, as shown in Lemma 13.

In our implementation, $Ordering$ constructs a priority queue Ω as a triple $(\Phi, \mathcal{T}, z)$ where Φ is an ordered list only for the itemsets of the highest priorities, $\mathcal{T} \supseteq \Phi$ additionally contains unordered itemsets that have relatively lower priorities than those in Φ, but have relatively higher priorities than the others, and z_μ for $\mu \in \mathcal{T}$ conceptually records how many subsets of μ are kept in K. Here, $\mathcal{T}$ is a candidate set for Φ, and every $\nu \in \mathcal{T}$ is contained in Φ if z_ν exceeds a certain threshold which will be specified in Sect. 4.1.1. Our $MostRel$ returns the itemset μ of the highest priority in $\Omega = (\Phi, \mathcal{T}, z)$, and updates Ω according to the change of irrelevancies of itemsets by the addition of μ to K.

4.1.1 Ordering

$Ordering$ has two tasks. First, it increases the counter of every itemset in $K \cap 2^I$ by 1. That is, all itemsets originally considered at Line 9 of Algorithm 3 are processed before the while loop. By Lemma 8, this preserves the correct processing order for 2^I.

Second, $Ordering$ constructs a priority queue Ω where a smaller irrelevancy implies a higher priority. The queue Ω keeps a correctly ordered list Φ only for itemsets with the highest priorities. Precisely, Φ contains every itemset $\mu \in 2^I$ such that $2 < f_K(\mu) \leq 3$ which means that all nonempty proper subsets of μ are already in K. Note that adding any itemset in Φ to K guarantees the **(P1)** property.

In addition, a set $\mathcal{T} \subseteq 2^I$ of itemsets, where $\Phi \subseteq \mathcal{T}$, with relatively lower priorities than Φ is kept unordered, which is a wait-list for addition to Φ. Precisely, the unordered set

$\mathcal{T}$ includes every itemset $\nu \in 2^I$ whose any subset with length $|\nu| - 1$ is in K, and every singleton of 2^I. Each itemset $\nu \in \mathcal{T}$ has an associated value z_ν recording the number of subsets of ν already contained in K, i.e. $|2^\nu \cap K|$. Instead of the exact number, we set z_ν to the number of such subsets whose length is at least $|\nu| - 1$, which enables to reduce time cost for updating z_ν. Note that by Lemma 7, $z_\nu = |\nu|$ implies that all nonempty proper subsets of ν are contained in K. This also means that $z_\mu = |\mu|$ for every $\mu \in \Phi$. Similarly, $z_\nu = |\nu| + 1$ implies $(2^\nu \setminus \emptyset) \subseteq K$. As a result, the priority queue Ω returned by $Ordering$ consists of the triple $(\Phi, \mathcal{T}, z)$.

To describe $\Omega = (\Phi, \mathcal{T}, z)$ more formally, we define the following two sets related to an itemset.

Definition 6 (1-*smaller Subset*) Given an itemset μ, its 1-smaller subsets $S(\mu)$ consists of subsets of μ whose length is smaller than μ by 1. That is,

$$S(\mu) = \{\nu \subset \mu : |\nu| = |\mu| - 1\}.$$

We especially denote the inclusion of μ to $S(\mu)$ by

$$S^+(\mu) = S(\mu) \cup \{\mu\}.$$

Definition 7 (1-*larger Superset*) Given an itemset μ and a transaction I, its 1-larger supersets $L_I(\mu)$ in I consist of supersets of μ in I whose length is larger than μ by 1. That is,

$$L_I(\mu) = \{\nu \subset I : \mu \subset \nu \text{ and } |\nu| = |\mu| + 1\}$$

We especially denote the inclusion of μ to $L_I(\mu)$ by

$$L_I^+(\mu) = L_I(\mu) \cup \{\mu\}.$$

Letting $\Psi = (2^I \cap K) \cup \emptyset$, the priority queue $\Omega = (\Phi, \mathcal{T}, z)$ is determined by $Ordering$ as follows.

$$\begin{aligned} \mathcal{T} &= \bigcup_{\mu \in \Psi} L_I^+(\mu), \\ z_\mu &= |S^+(\mu) \cap K|, \ \forall \mu \in \mathcal{T}, \\ \Phi &= \{\mu \in \mathcal{T} : z_\mu = |\mu|\}. \end{aligned} \tag{1}$$

Algorithm 4 presents the whole procedure of $Ordering$. In Line 1, $Scan(K, I)$ performs the first task. It computes $\Psi = K \cap 2^I$, which is sorted in the descending order of their lengths, while increasing c_μ by 1 for every $\mu \in \Psi$.

After appending $\emptyset$ to Ψ, the second task is performed, i.e. Φ, $\mathcal{T}$, and z are constructed. Every $\mu \in \Psi$ is added to $\mathcal{T}$ with $z_\mu = |\mu| + 1$ since $\mu \in K$. For each $\nu \in L_I(\mu)$, if $\nu \in \mathcal{T}$, we set $z_\nu = \min(z_\nu + 1, |\nu| + 1)$: i.e. if ν is an element of Ψ, $z_\nu = |\nu| + 1$ remains unchanged. If $\nu \notin \mathcal{T}$, ν is added to $\mathcal{T}$ with $z_\nu = 1$. After that, if z_ν becomes equal to $|\nu|$, ν is added to the tail of Φ.

Note that itemsets in Φ are sorted in the ascending order of their irrelevancy scores f_K. This is because 1) $|2^\nu \setminus K| = 2$ by $z_\mu = |\mu|$ for every $\mu \in \Phi$, and 2) Φ is sorted in the descending order of their lengths due to Ψ considered in the descending order of itemset lengths at Line 4 of Algorithm 4. Also note that $f_K(\mu) < f_K(\nu)$ for $\mu \in \Phi$ and $\nu \in 2^I \setminus K \setminus \Phi$ because all nonempty proper subsets of μ are in K, but not for ν, i.e. $f_K(\mu) \leq 3$ and $f_K(\nu) > 3$.

Figure 2 shows an example of $\Omega = (\Phi, \mathcal{T}, z)$ constructed by $Ordering$ for a given K and I. The leftmost box corresponds to the current K and I; the middle corresponds to Ω

Algorithm 4: Implementation of *Ordering*

Input: A transaction I, and a set K of itemsets currently counted.
Output: A priority queue $\Omega = (\mathcal{T}, \Phi, z)$.

1 $\Psi \leftarrow Scan(K, I)$.
2 Append $\emptyset$ to Ψ. `// Ψ is a double ended queue`
3 $\mathcal{T} \leftarrow \Phi \leftarrow \emptyset$.
4 **for** *every* $\mu \in \Psi$ *in the order* **do**
5     $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mu\}$.
6     $z_\mu \leftarrow |\mu| + 1$.
7   **for** *every* $\nu \in L_I(\mu)$ **do**
8   **if** $\nu \in \mathcal{T}$ **then**
9             $z_\nu \leftarrow \min(z_\nu + 1, \ |\nu| + 1)$.
10   **else**
11             $\mathcal{T} \leftarrow \mathcal{T} \cup \{\nu\}$ with $z_\nu \leftarrow 1$.
12   **end**
13   **if** $z_\nu = |\nu|$ **then**
14             $\Phi.push_tail(\nu)$.
15   **end**
16   **end**
17 **end**

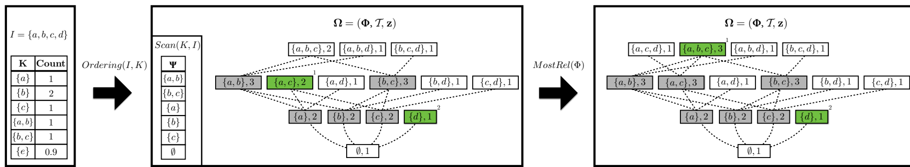


Fig. 2 Example of $\Omega = (\Phi, \mathcal{T}, z)$ and its change by *MostRel*. The *boxes* correspond to $\mathcal{T}$ where the *green* ones correspond to Φ and the gray ones contain itemsets already in K. Each *box* shows the corresponding itemset μ and z_μ. The attached number to a *green box* denotes the priority in Φ where a smaller number means a higher priority. The dotted line denotes set inclusion relation for two sets; note that for the itemset μ of each non-gray box, z_μ is equal to the number of *dotted lines* attached to the *bottom* of the *box*. Calling *MostRel* returns $\{a, c\}$ with the highest priority and updates Ω as in the rightmost figure. After that, Algorithm 4 adds $\{a, c\}$ to K and evicts $\{e\}$ from K since $\{e\}$ is the only element that neither has the count at least 1 nor is an element of 2^I, i.e. $\{e\}$ is the only element of $\Theta = Faded(K, I)$. This completes the iteration since Θ becomes empty (color figure online)

after calling $Ordering(I, K)$. The itemset and the number in each box are an element of $\mathcal{T}$ and the corresponding z value, respectively. The itemset in each gray box corresponds to $\mu \in 2^I \cap K$ for which the associated count c_μ increases by 1 in $Scan(K, I)$. The green boxes correspond to Φ, and the priority of the itemset is attached to each box where a smaller number implies a higher priority. Note that all nonempty proper subsets of the itemset in each green box are already in K, and $\{a, c\}$ has a higher priority than $\{d\}$ because $|\{a, c\}| > |\{d\}|$.

As a result, Φ consists of itemsets of the highest priorities with the correct order.

Lemma 11 *The time complexity of Algorithm 4 is* $O\left(|K|\left(|I| + \log|K|\right)\right)$ *with a constant itemset size.*

Proof Line 1 requires $O(|K|\log|K|)$. First, computing $\hat{\Phi} = K \cap 2^I$ takes $O(|K|)$ time since each element $\mu \in K$ is verified if it is a subset of I or not by checking each $u \in I$ for every $u' \in \mu$. Second, sorting $\hat{\Phi}$ takes $O(|K|\log|K|)$ since $|\hat{\Phi}| \leq |K|$.

For every set μ, $|L_I(\mu)| \leq |I|$ by definition. Thus, the outer and inner for loops run at most $|K| \geq |\Phi|$ and $|I| \geq |L_I(\mu)|$ iterations, respectively. As a result, Algorithm 4 has $O\left(|K|\left(|I| + \log|K|\right)\right)$ time complexity. □

4.1.2 MostRel

Next, we explain $MostRel$ in detail which returns the itemset μ of the highest priority in $\Omega = (\Phi, \mathcal{T}, z)$. Finding μ is trivial because Φ is sorted in the ascending order of f_K, i.e. the desired itemset is at the head of Φ. As shown in Algorithm 3, μ returned by $MostRel$ is added to K at Line 12, or the iteration terminates. Note that μ never enters Line 9 of Algorithm 3 since the task in Line 8–9 is already processed in $Scan$ of Algorithm 4. In this scenario, there are two problems: 1) Φ can be empty after removing μ, and 2) Φ can be out of order due to the addition of μ to K. Thus, we need to update Ω appropriately for the next use.

Before adding μ to K, Φ contains the top-$|\Phi|$ itemsets in the correct order, and z_ν for $\nu \in \Phi$ is unchanged unless ν itself is added to K. In other words, the relative order of itemsets in Φ remains the same. Thus, we only need to focus on itemsets newly added to Φ by the addition of μ to K. Candidates for such itemsets are those ν for which z_ν increases by 1. Considering these candidates, the update is done for preserving the definitions of Φ, $\mathcal{T}$ and z given by Eq. (1) after the addition of μ to K.

Algorithm 5: Implementation of $MostRel$

Input: A priority queue $\Omega = (\mathcal{T}, \Phi, z)$, and a transaction I.
Output: The most relevance itemset μ.

```
1 μ ← Φ.pop().
2 for every ν ∈ L_I(μ) do
3   if ν ∈ T then z_ν ← z_ν + 1.
4   else T ← T ∪ {ν} with z_ν ← 1.
5   if z_ν = |ν| then
6     Φ.push_head(ν).
7   end
8 end
9 z_μ ← z_μ + 1.
```

Algorithm 5 describes the update procedure. In the algorithm, $\Phi.pop()$ returns the head itemset of Φ and remove it. In Line 2, every ν whose z_ν changes is examined. If ν is already in $\mathcal{T}$, its z_ν increases by 1; otherwise ν is added to $\mathcal{T}$ with $z_\nu = 1$. After that if $z_\nu = |\nu|$, ν is placed at the head of Φ, i.e. ν has the highest priority. This addition guarantees the correct order because for every itemset $\omega \in \Phi$, $z_\omega = |\omega|$ and $|\nu| > |\mu| \geq |\omega|$.

Figure 2 shows the change of $\Omega = (\Phi, \mathcal{T}, z)$ by $MostRel$ in the rightmost figure. The itemset $\{a, c\}$ of the highest priority before calling $MostRel$ is popped from Φ. Note that the update is done for the case $\{a, c\}$ is added to K so that the box of $\{a, c\}$ is colored in gray. Accordingly, $z_{\{a,b,c\}}$ where $\{a, b, c\} \in L_I(\{a, c\})$ increases by 1 which makes $\{a, b, c\}$ added to Φ because $z_{\{a,b,c\}} = |\{a, b, c\}| = 3$. Note that $\{a, b, c\}$ has a higher priority than $\{d\}$ since $\{a, b, c\}$ is added at the head of Φ, which matches $|\{a, b, c\}| > |\{d\}|$. Another element $\{a, c, d\} \in L_I(\{a, c\})$ is newly added to $\mathcal{T}$ with $z_{\{a,c,d\}} = 1$.

Lemma 12 *The time complexity of Algorithm 5 is $O(|I|)$ with a constant itemset size.*

Proof The time consumption is determined by the for loop, and $L_I(\mu) \leq |I|$ for every set μ. □

4.1.3 Time costs of TwMinSwap- Is *and naive method*

Below, we analyze the time complexity of TwMinSwap- Is.

Lemma 13 *The time complexity of* TwMinSwap- Is *for every new transaction is* $O(|K|(|I| + \log|K|))$ *with a constant itemset size.*

Proof The only remaining time-consuming part is the while loop in Line 6 of Algorithm 3. Line 8 of Algorithm 3 is never reached in our implementation, and $|\Theta| \leq |K|$. Hence, the number of iterations by the while loop is $O(|K|)$. Combining Lemmas 11 and 12, the processing for a new transaction takes $O\left(|K|\left(|I| + \log|K|\right)\right)$. □

Comparing with our implementation, the naive approach described at the beginning of Sect. 4.1 is greatly degraded. It requires $O(1)$ and $O(|2^I||K|)$ times for $Ordering$ and $MostRel$, respectively, since for every $\mu \in 2^I$, $f_K(\mu)$ should be calculated whenever $MostRel$ is called. More worse is that $MostRel$ is called at most K times while $Ordering$ is called only one time for every new transaction. Consequently, the naive approach takes $O(|2^I||K|^2)$ time for each transaction. Consequently, TwMinSwap- Is is orders of magnitude faster than the naive method.

5 Experiments on frequent item mining

In this section, we present experimental results to show the performance of our proposed TwMinSwap with synthetic data streams. Especially, we want to answer the following questions:

Q1 How many top-k time-weighted frequent items can we discover?
Q2 How accurately can we estimate time-weighted counts of the discovered items?
Q3 How fast is TwMinSwap?

5.1 Setup

We consider two types of data streams generated from power-law distributions to simulate bursty item occurrences where N is the stream length and n is the number of distinct items.

– Static distribution: N size-one transactions are generated from the following power-law distribution.

$$\Pr\left[i \text{ is generated}\right] \propto i^{-\beta},$$

where $i \in [1, n]$.

– Dynamic distribution: for $0 \leq r \leq 1$, the first rN size-one transactions are generated from

$$\Pr\left[i \text{ is generated}\right] \propto i^{-\beta},$$

and the last $(1 - r)N$ size-one transactions are generated from

$$\Pr\left[i \text{ is generated}\right] \propto (n - i + 1)^{-\beta},$$

where $i \in [1, n]$. This provides the setting that frequent items differ from time-weighted frequent items.

Table 4 Top-5 items with and without time-weighting for the two types of item distributions

	Rank	No time-weighting		Time-weighting	
		Static	Dynamic	Static	Dynamic
(High)	1	1	1	1	10000
	2	2	2	2	9999
	3	3	3	5	9998
	4	4	10000	4	9995
(Low)	5	5	4	7	9997

With the dynamic item distribution, frequent items are different from those for the static distribution. Especially, with time-weighting, frequent items are completely different between the static and the dynamic distributions—items *recently* occurring many times are placed in high ranks for the dynamic distribution

Table 4 shows the difference between the two types of distributions.

In our experiments, β is varied in $\{0.5, 0.75, 1, 1.25, 1.5, 1.75\}$; $N = 10^6$, $n = 10^4$, $r = 0.8$ and $k = 50$ are fixed. Here, as β gets larger, item frequencies get skewed more. Note that although the stream length N is fixed in our experiments, the per-item time and the space complexities of our algorithms are independent on N.

We consider the following competitors, and in our experiments, all methods are implemented in Java.

– TwFreq [39]: a counter-based algorithm to find time-weighted frequent items from a data stream.
– TwHCount [9]: a sketch-based algorithm to find time-weighted frequent items from a data stream.
– SpaceSaving [10,35]: a counter-based algorithm to find frequent items from a data stream. Since this is without time-weighting, we compare this algorithm with the others only in precision and recall.

The memory requirements for all the algorithms are as follows. TwMinSwap, TwFreq, and SpaceSaving require $O(k)$ memory spaces; TwSample requires $O(sk)$ where s is the number of parallel sessions each of which independently samples at most k distinct items; TwHCount requires $O(k + rm)$ where r is the number of hash functions and m is a range size of the hash functions.

We denote TwSample with s number of the independent sessions by TwSample(s), and TwHCount with the hash table size $w\%$ of n, i.e. $rm = \frac{wn}{100}$, by TwHCount(w). For TwSample, we use $s = 10$ and $\sigma = 0.0001$; for TwSample and TwMinSwap, we use $\alpha = 0.99$; for TwHCount, we use the parameters in the original paper [9], and set hash table sizes rm to 1 and 10% of n.

The overall comparison is summarized in Table 1 and Fig. 1. TwMinSwap outperforms the others in terms of accuracy and memory usage, and its speed is comparable to that of the fastest competitor TwHCount.

5.2 Discovering top-k time-weighted frequent items

Figure 3 presents accuracy of discovered items in terms of precision and recall defined as follows:

$$\text{precision} = \frac{|\Theta \cap \hat{\Theta}|}{|\hat{\Theta}|}, \text{ and recall} = \frac{|\Theta \cap \hat{\Theta}|}{|\Theta|}, \tag{2}$$

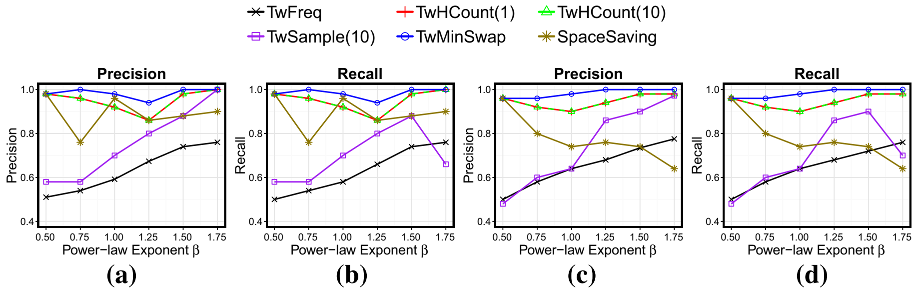


Fig. 3 Our proposed TwMinSwap shows the best performance in both precision and recall whose values are close to 1, regardless of the distribution types and their β values. For distributions with small β in which frequencies of items are relatively even, TwSample shows low precision and recall, but as β gets larger, the values rapidly increase. TwFreq also shows better performance for large β, but the improved precision and recall are limited below 0.8. Regardless of the hash table size, TwHCount shows the second best performance; however, the performance is below that of TwMinSwap. As expected, SpaceSaving is degraded for the dynamic item distributions while performing well on average for the static item distributions. **a** Static distribution, **b** static distribution, **c** dynamic distribution, **d** dynamic distribution

where Θ and $\hat{\Theta}$ are the sets of the true and estimated top-k items, respectively. Overall, our proposed TwMinSwap outperforms other algorithms regardless of the types of item distributions and the exponent values β. Its precision and recall are always very close to 1. TwSample(10) improves precision and recall as β gets larger in general. The reason why the recall of TwSample(10) is low for $\beta = 1.75$ is that a large amount of probability density is assigned to only fewer items as β gets larger, leading to a small number $\hat{k} < k$ of discovered items. The exact numbers are 33 and 36 for the static and the dynamic distributions, respectively. For TwMinSwap and TwHCount, always $\hat{k} = 50$, and for TwFreq, always $\hat{k} \geq 49$.

TwFreq shows a similar pattern to TwSample(10), but its improvement is less significant than TwSample(10). TwHCount results in high precision and recall regardless of hash table sizes, but they are still below TwMinSwap. SpaceSaving performs quite well for the static item distributions: both precision and recall are about 0.9 on average. However, since SpaceSaving does not consider a time-weighting factor, for the dynamic item distributions its performance is greatly degraded as β gets larger.

The overall result implies that our time-weighted counting plays an important role in finding recent frequent items from data streams.

5.3 Estimating time-weighted counts

Accurately estimating time-weighted counts for discovered items enables to quantitatively compare them. Figure 4 shows estimated time-weighted counts of the discovered top-k items. The error ϵ_i for $1 \leq i \leq k$ is calculated as follows:

$$\epsilon_i = \frac{|x_i - \hat{x}_i|}{x_i}, \tag{3}$$

where x_i and $\hat{x}_i$ are the true and estimated time weighted counts of the top-ith item, respectively. Notably, TwMinSwap estimates the true time-weighted counts very accurately, which is shown by the blue line (almost overlapped with green).

Since TwFreq provides an upper and a lower bounds of the true time-weighted count, we choose the mean of the two bounds. In general, as ranks of items get lower, its accu-

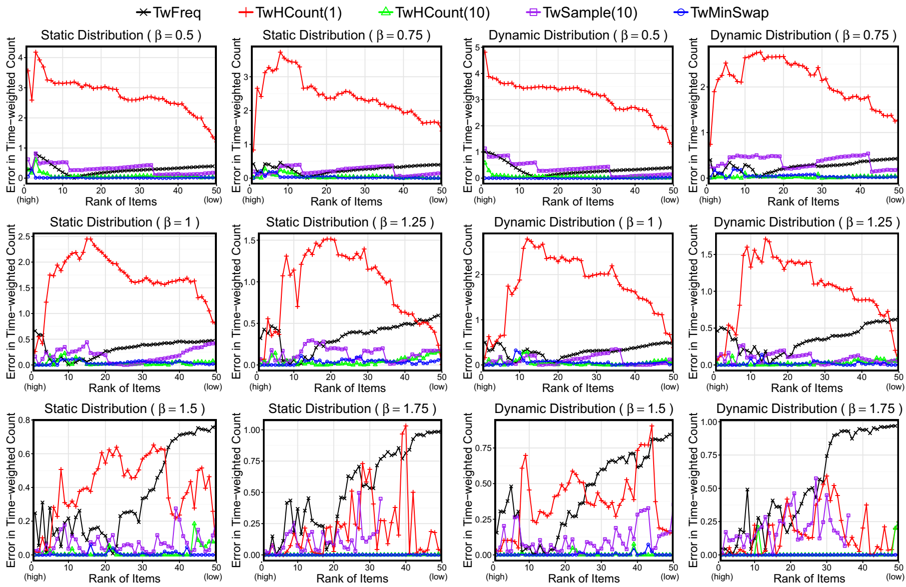


Fig. 4 The estimated time-weighted counts by TwMinSwap are the most accurate. The *plots* show the error in estimated time-weighted counts of the algorithms for at most k number of discovered items (the lower, the better). While TwHCount(10) shows the second best performance, TwHCount(1) using smaller memory spaces performs poorly especially for small β. The accuracy of TwFreq is generally better for high rank items than for low rank items. TwSample(10) shows slightly better performance than TwFreq on average, but in contrast to TwFreq, there is no great degradation of the estimations for low rank items in TwSample(10)

racy is generally degraded. One reason of the poor estimation of TwFreq for low rank items is that TwFreq is originally developed to find frequent items having time-weighted counts above a certain threshold rather than top-k ones though it maintains at most k items. TwSample(10) shows the third best performance and estimates time-weighted counts more accurately for relatively large β. The performance is slightly better than TwFreq on average, but TwSample(10) is not degraded for relatively low rank items.

The performance of TwHCount highly depends on the hash table size rm. The estimation of TwHCount(1) results in larger error while TwHCount(10) is comparable to TwMinSwap. This is notable because the estimation of TwMinSwap is as accurate as that of TwHCount(10) which keeps an additional hash table of size $0.1n$ for accuracy. The comparison of the error on average is shown in Fig. 1.

5.4 Running time

Figure 5 shows running times of the algorithms taken to process all the items in the data streams. The overall trend is that running time decreases over increasing β. This is because with large β, the probability of a new item being already monitored increases, leading to infrequent invoking eviction processes such as *DownSampling* in TwSample. TwSample(10) shows the slow running times due to performing 10 independent sampling.

Although TwFreq has the same per-item processing time as TwMinSwap in the big-O notation, TwFreq involves more computations for the eviction process than TwMinSwap. TwFreq updates more variables and scans monitored items twice while TwMinSwap scans

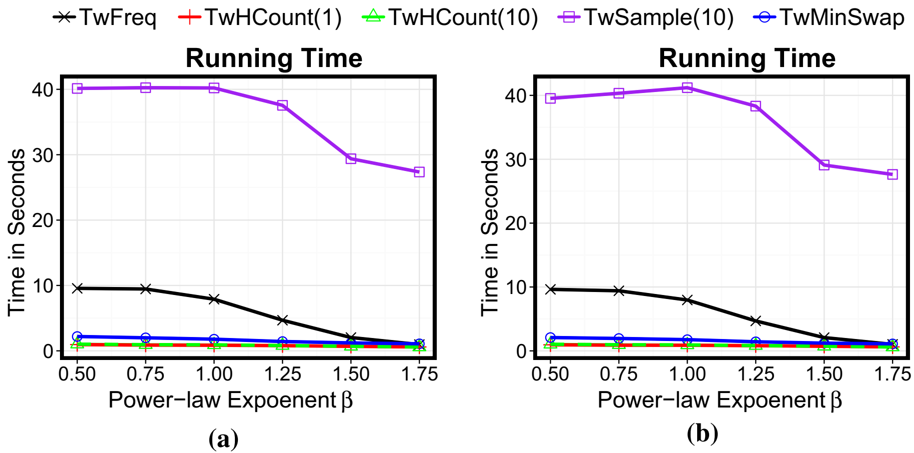


Fig. 5 TwMinSwap is faster than TwSample(10) and TwFreq in most cases of β. Although TwHCount is slightly faster than TwMinSwap, its memory usage is much larger than TwMinSwap as shown in Fig. 1. **a** Static distribution, **b** dynamic distribution

them once. As a result, the running time of TwFreq changes more dramatically than TwMinSwap.

Since TwHCount has $O(r)$ per-item processing time, it is the fastest. In fact, this fast running time is achieved by maintaining the hash table of size rm for approximate time-weighted counting, which leads to more memory spaces than TwMinSwap. Although the running time of TwHCount does not depend on m, to guarantee small error of estimated time-weighted counts of discovered items, rm should be large as shown in Fig. 4. In our experiments, $rm = 0.1n$ is satisfactory while $rm = 0.01n$ is not.

All algorithms show no meaningful difference in running time with respect to the two types of distributions.

5.5 Effect of parameters α and k

We investigate the two parameters of TwMinSwap: the time-decaying factor α and the number k of time-weighted frequent items to be discovered. Figure 6 shows precision and mean of error of TwMinSwap while changing α and k for a dynamic item stream with $\beta = 1$. TwMinSwap works very accurately for $\alpha \in \{0.9, 0.95, 0.99\}$. For extremely large $\alpha = 0.999$, TwMinSwap is relatively degraded with a small k, but still keeps the precision of about 0.7 and the error of about 0.2.

6 Experiments on frequent itemset mining

In this section, we present experimental results to show the performance of our proposed TwMinSwap- Is.

6.1 Setup

Table 5 lists the datasets used in our experiments. Figure 7a shows the cumulative size distribution of transactions for every dataset. Note that a large portion of transactions have a short length. We consider the following competitor:

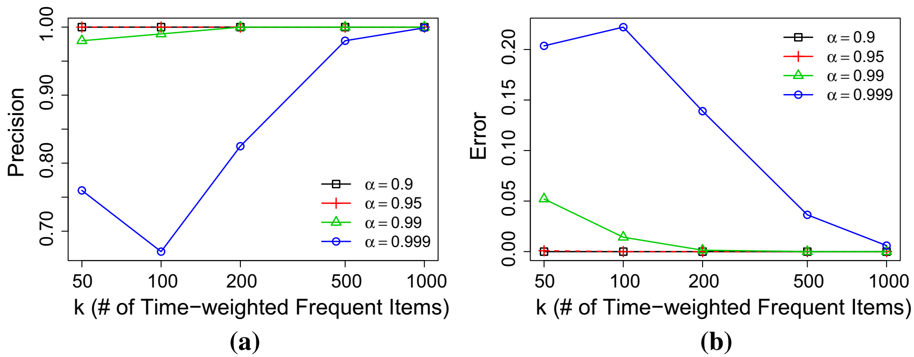


Fig. 6 Precision and error of TWMINSWAP with changing k and α. A dynamic stream with $\beta = 1$ is used. TWMINSWAP outputs accurate results unless α becomes too large. Even for extremely large $\alpha = 0.999$, the accuracy is maintained as about 0.7 and 0.2 for the precision and the error, respectively, when k is small, and improved as k gets larger. **a** Precision, **b** error

Table 5 Dataset used in our experiments on frequent itemset mining

Name	# of Trans.	Max. Trans. Len.	Description
Retail [4]	88,162	76	From retail market
BMS1[a]	59,602	267	Click stream
BMS2[a]	77,512	161	Click stream

[a] http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php

- SKIP LC- SS [38]: a method to find frequent itemsets in data streams which is based on combining LOSSYCOUNTING and SPACESAVING.

Although SKIP LC- SS does not consider time-weighting, we choose it as a competitor to examine the effect of the Apriori property preserved in the top-k results: SKIP LC- SS maintains at most k itemsets but neglects the Apriori property in them. For all experiments, we use $k = 1000$ and $\alpha = 0.999$. We omit evaluation using precision and recall since calculating the true time-weighted counts for all itemsets is intractable: e.g., there are $2^{267} - 1$ itemsets for the largest transaction of BMS1.

6.2 Results

Figure 7b shows the true time-weighted count and the true count of itemsets for Retail found by TWMINSWAP- IS and SKIP LC- SS over their ranks, respectively. TWMINSWAP- IS ranks itemsets according to their true time-weighted counts, while SKIP LC- SS does not perform well.[4] The main reason for the poor performance of SKIP LC- SS is that it swaps itemsets frequently since it randomly selects itemsets to be monitored among all subsets of a transaction. In addition, we observe that TWMINSWAP- IS detects recently frequent itemsets despite its small occurrences in total. For instance, in Retail, the two itemsets of $\mu = \{39, 48, 4994\}$ and $\nu = \{36, 16430, 16431\}$ occur 129 and 9 in the stream, respectively. At the end of the stream, however, only ν is identified as one of the top-k itemsets by TWMINSWAP- IS, because ν occurs more recently than μ as shown in Fig. 8.

[4] In the original paper proposing SKIP LC- SS, k is set to a large $50{,}000 \leq k \leq 70{,}000$.

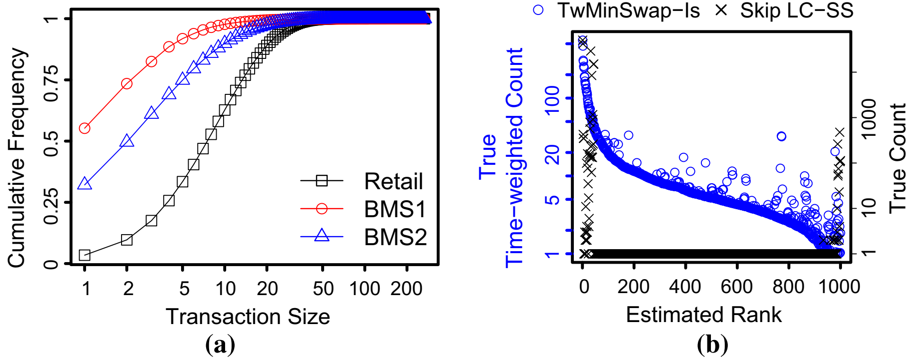


Fig. 7 **a** Cumulative distribution on transaction sizes for every dataset. **b** True time-weighted count (*left* y-axis) and true count (*right* y-axis) for itemsets discovered by TwMinSwap- Is and Skip LC- SS over their corresponding estimated ranks (x-axis), respectively, for Retail. Note that the itemsets by TwMinSwap- Is are approximately ranked with respect to their time-weighted counts, while it is not the case for Skip LC- SS

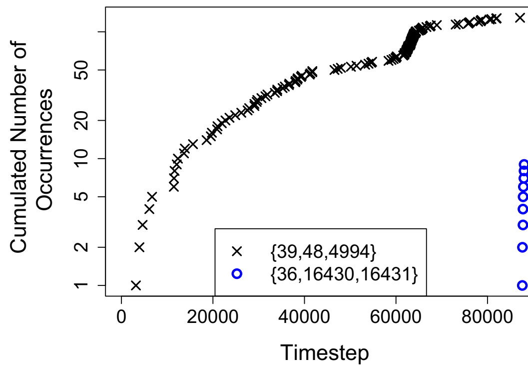


Fig. 8 In Retail, itemset {36, 16430, 16431} occurs only 9 times but very recently, which leads to it detected by TwMinSwap- Is. In contrast, much frequent one {39, 48, 4994} with 129 occurrences over the entire stream is not identified as a time-weighted frequent itemset by TwMinSwap- Is since it is rare in recent times. We omit points for which the corresponding y-axis value is 0 or remains the same as the previous one

Figure 9a shows average error in time-weighted count over all itemsets discovered by TwMinSwap- Is. During the entire stream, TwMinSwap- Is maintains small error in time-weighted counts on average. Figure 9b shows the size distribution of itemsets found by TwMinSwap- Is for every dataset. The largest itemset size is about 5–7, and most of transactions in the datasets are shorter than the size as shown in Fig. 7a. In other words, an itemset whose length is larger than 7 has a low chance to be frequent in the time-weighted count in nature.

7 Discovery

In this section, we present discoveries from applying our methods to several real-world data streams. For each dataset, the used time decaying factor α is specified. Although choosing the optimal α is not trivial, we observed that in practice, TwMinSwap and TwMinSwap- Is work well with a large $\alpha \geq 0.9$.

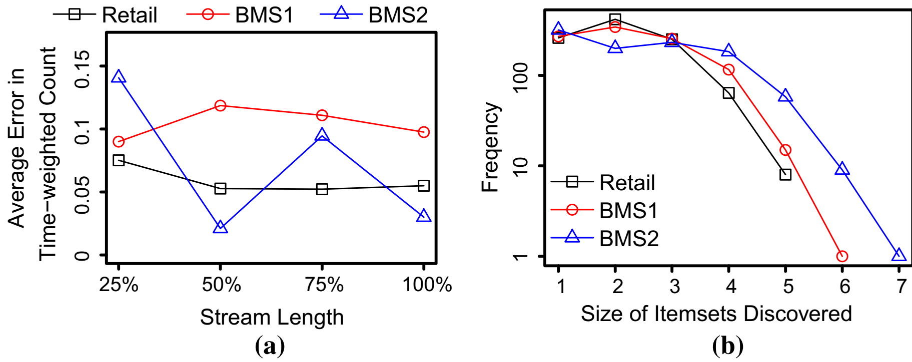


Fig. 9 **a** Error in time-weighted count for itemsets found by TwMinSwap- Is for Retail. Note that while processing the stream, the error is kept low for all datasets. **b** Size distribution of itemsets found by TwMinSwap- Is. Itemsets of a moderate size (5–7) are discovered for every dataset

7.1 MemeTracker dataset

Setup The MemeTracker dataset [24] provides quotes and phrases from blogs and news media. We consider a keyword stream consisting of words in the quotes and the phrases where 571 stopwords provided in [25] are excluded. The stream covers time period between August 2008 and April 2009; the length of the stream is 1,681,760,809; we consider 1 min as one time step.

Results We run TwMinSwap with $k = 300$ and $\alpha = 0.9$, and examine top-300 keywords for every month.

Figure 10a shows the tracking results of keywords related to the U.S. presidential election in Nov 4 2008. The values for each month are normalized time-weighted counts divided by the sum of those of k number of discovered items. Since multiple items can occur at one time step, this normalization is required to eliminate effects of undesirably large time-weighted counts due to relatively large stream lengths for certain time periods. Both keywords related to the candidates *obama* and *mccain* were mentioned actively before, and received less attentions after the election. Notably, despite high frequencies of both keywords, the winner *obama* was more frequently mentioned in blogs and media than the loser *mccain*. Even after the election, *obama* occasionally becomes hot.

Figure 10b, c show sudden arising and quick vanishing of keywords closely related to two incidents: the Mumbai terror attack in Nov 2008, and the Gaza War beginning on Dec 2008. Although each incident happened in the last part of a month, TwMinSwap correctly detects related keywords as hot items in the report for that month.

7.2 Amazon movie review dataset

Setup The Amazon movie review dataset [34] provides user reviews with product id information where the movie title of each product id can be checked by http://www.amazon.com/dp/PRODUCT_ID. We consider the stream of the product ids. The stream covers the period from Aug 20 1997 to Sep 25 2012; the length of the stream is 7,911,684; the number of distinct product ids is 253,059.

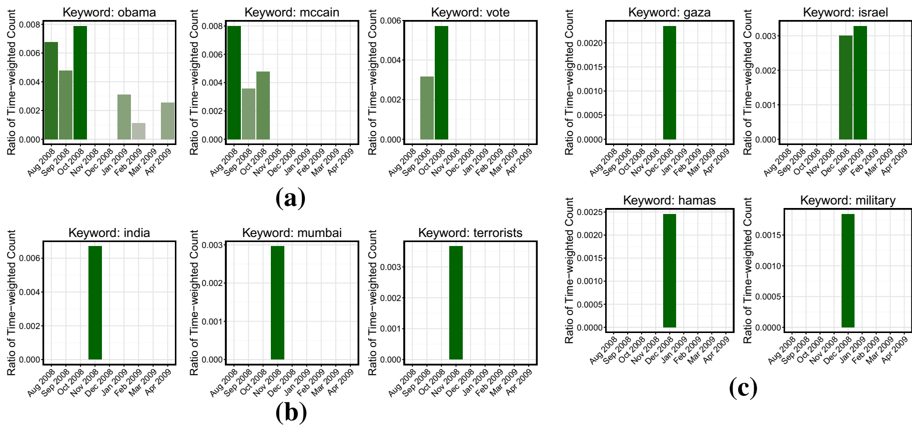


Fig. 10 **a** Changes of time-weighted counts of keywords related to the presidential election in Nov 4 2008. The keywords *obama, mccain* and *vote* become hot keywords before and cooled down after the election. Notably, the winner *obama* is more actively mentioned than the loser *mccain* before the election. Also, the winner *obama* does not disappear after the election in contrast to the loser *mccain*. **b** Changes of time-weighted counts of keywords related to the Mumbai terror attack in Nov 26 2008. As similar to the pattern for the Gaza War, keywords related to the Mumbai terror attack such as *india, mumbai* and *terrorists* show sudden rises right after the attack time. **c** Changes of time-weighted counts of keywords related to the Gaza War beginning in the last part of Dec 2008. In December, several keywords related to the war suddenly had arisen and quickly disappeared. Note that the war ended in the middle part of January

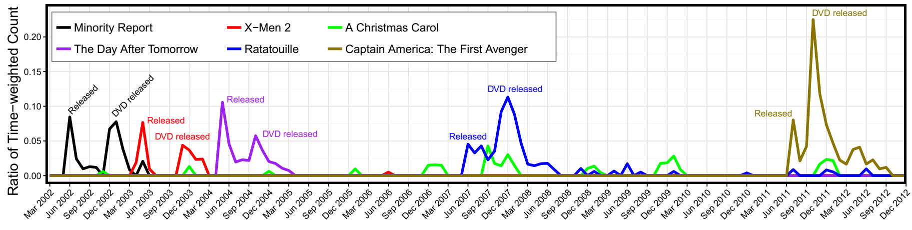


Fig. 11 Changes of time-weighted counts of movies reviewed by users in Amazon. We observe two patterns. (1) The general pattern is doubly-active attention to movies when they are released at theaters and in DVDs. (2) The minor pattern is periodical attention: e.g., *A Christmas Carol* is popular in every winter

Results We run TwMinSwap with $k = 100$ and $\alpha = 0.9$. Figure 11 shows the tracking result of several movies among the top-100, which is summarized as two patterns. The major pattern is doubly-active attention when a movie is released at theaters and in DVD as for *Minority Report, X-Men 2, The Day After Tomorrow, Ratatouille*, and *Captain America: The First Avenger*. The other pattern is periodical attention: e.g. *A Christmas Carol* appears in every winter.

7.3 Yelp dataset

Setup The Yelp dataset[5] provides tip data for businesses by users. Here, the tips are short comments about the businesses, and each business has several associated categories. We consider the stream of the business ids. The stream covers the period from April 16 2009 to

5 http://www.yelp.com/dataset_challenge/.

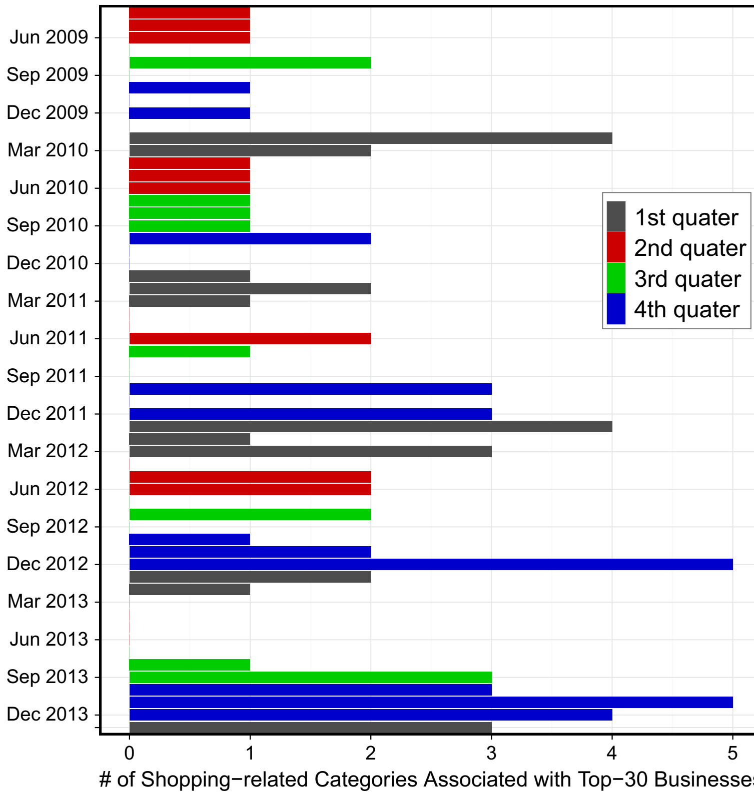


Fig. 12 Count distribution over time for shopping-related categories—*Shopping* and *Shopping Centers*. For each month, the value is calculated with respect to top-30 businesses. In winter, there were a number of visits to shopping businesses

February 11 2014; the length of the stream is $113,993$; the number of distinct businesses is $15,585$.

Results We run TwMinSwap with $k = 50$ and $\alpha = 0.9$, and track the top-30 hot businesses per month. For each month, we obtain a category distribution with respect to the top-30 businesses. Figure 12 shows the result for shopping-related categories—*Shopping* and *Shopping Centers*. Around the new years, shopping activity increased. This reflects that there are several special days such as Christmas, New Year, and Valentine Day in winter.

7.4 Human contact dataset

Setup The Human Contact dataset,[6] originally provided in [15], contains physical human contact information over 9 months. The dataset is given as an undirected multigraph with timestamps for edges where a node and an edge correspond to a person and its contact, respectively. To make it transactional data, we (1) divide time into non-overlapping intervals of 10 min, (2) construct a subgraph with nodes and edges appearing in each interval, and

[6] http://konect.uni-koblenz.de/networks/mit.

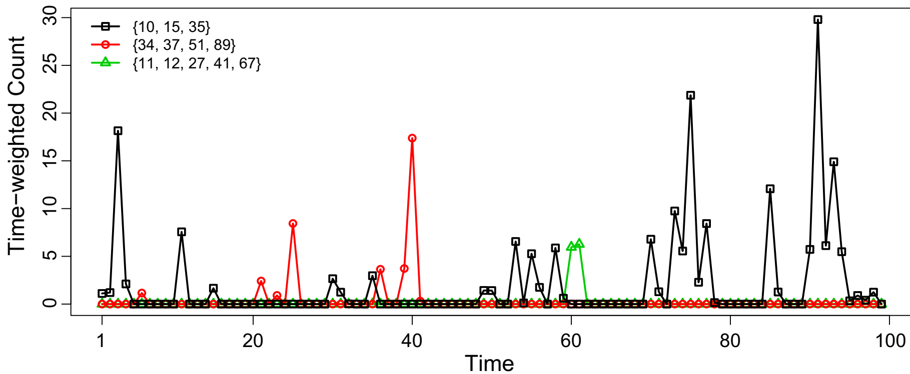


Fig. 13 Time-weighted counts of several itemsets over time. We observe three patterns in physical human contacts: periodical (*black*), occasional (*red*), and rare (*green*) (color figure online)

(3) for each node in the subgraph, make one transaction consisting of all neighbors of the person corresponding to the node. The resulting stream contains 94 persons and 239, 443 transactions.

Results We run TwMinSwap- Is with $k = 300$ and $\alpha = 0.998$. We divide the 9 months into 100 equal-sized time intervals, and track the top-300 time-weighted frequent itemsets at every interval boundary. Figure 13 shows changes of time-weighted counts for several itemsets, which exhibit different contact patterns, over time. For instance, the three persons 10, 15 and 35 in black meet a number of common people periodically. Such meetings are occasional for the persons 34, 37, 51 and 89 in red, and rare for 11, 12, 27, 41 and 67 in green. This shows the effectiveness of TwMinSwap- Is in discovering temporal patterns in stream data.

8 Conclusion

In this paper we propose algorithms to track recently frequent patterns in high-speed data streams: TwMinSwap for items and TwMinSwap- Is for itemsets. We propose TwMinSwap for efficient time-weighted counting of *items* in data streams. TwMinSwap is inspired by TwSample, our sampling-based randomized algorithm with theoretical guarantees. Both methods require only $O(k)$ memory spaces for tracking top-k items. We also propose TwMinSwap- Is for time-weighted *itemset* counting. TwMinSwap- Is guarantees self-consistency in results, i.e., the Apriori property holds, and requires $O(k)$ memory spaces for a constant itemset size. Conducting extensive experiments on synthetic and real data streams, we show that TwMinSwap is fast and outperforms all existing methods in accuracy; TwMinSwap- Is is accurate and discovers itemsets of a non-trivial length. Analyzing real-world data streams, we discover interesting patterns, including the difference of trends between the winner and the loser of the U.S. presidential election, and temporal patterns in physical human contacts.

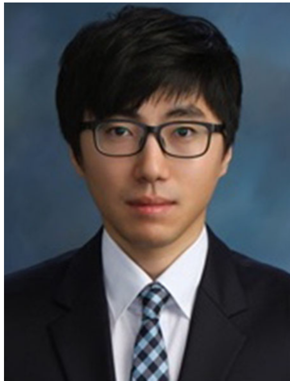
Acknowledgements This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-15-2012, High Performance Big

Data Analytics Platform Performance Acceleration Technologies Development). The Institute of Engineering Research at Seoul National University provided research facilities for this work. The ICT at Seoul National University provides research facilities for this study.

References

1. Arasu A, Manku GS (2004) Approximate counts and quantiles over sliding windows. In: PODS
2. Borgelt C, Yang X, Nogales-Cadenas R, Carmona-Saez P, Pascual-Montano A (2011) Finding closed frequent item sets by intersecting transactions. In: EDBT
3. Boyer RS, Moore JS (1991) Mjrty: a fast majority vote algorithm. In: Automated reasoning: essays in honor of Woody Bledsoe
4. Brijs T, Swinnen G, Vanhoof K, Wets G (1999) Using association rules for product assortment decisions: a case study. In: Knowledge discovery and data mining
5. Calders T, Dexters N, Gillis JJM, Goethals B (2014) Mining frequent itemsets in a stream. Inf Syst 39:233–255
6. Chang JH, Lee WS (2003) Finding recent frequent itemsets adaptively over online data streams. In: KDD
7. Chang J H, Lee W S (2004) Decaying obsolete information in finding recent frequent itemsets over data streams. IEICE Trans 87–D(6):1588–1592
8. Charikar M, Chen K, Farach-Colton M (2002) Finding frequent items in data streams. In: ICALP
9. Chen L, Mei Q (2014) Mining frequent items in data stream using time fading model. Inf Sci 257:54–69
10. Cormode G, Hadjieleftheriou M (2010) Methods for finding frequent items in data streams. VLDB J 19(1):3–20
11. Cormode G, Muthukrishnan S (2004a) An improved data stream summary: the count-min sketch and its applications. In: LATIN
12. Cormode G, Muthukrishnan S (2004b) What's new: finding significant differences in network data streams. In: INFOCOM
13. Dallachiesa M, Palpanas T (2013) Identifying streaming frequent items in ad hoc time windows. Data Knowl Eng 87:66–90
14. Demaine ED, Lpez-Ortiz A, Munro JI (2002) Frequency estimation of internet packet streams with limited space. In: ESA
15. Eagle N, Pentland A (2006) Reality mining: sensing complex social systems. Pers Ubiquitous Comput 10(4):255–268
16. Fischer MJ, Salzberg SL (1982) Finding a majority among n votes: solution to problem 81–5 (Journal of Algorithms, June 1981). J Algorithms 3(4):362–380
17. Gibbons PB, Matias Y (1998) New sampling-based summary statistics for improving approximate query answers. In: SIGMOD
18. Golab L, DeHaan D, Demaine ED, López-Ortiz A, Munro JI (2003) Identifying frequent items in sliding windows over on-line packet streams. In: Internet measurement conference
19. Golab L, DeHaan D, López-Ortiz A, Demaine ED (2004) Finding frequent items in sliding windows with multinomially-distributed item frequencies. In: SSDBM
20. Jin C, Qian W, Sha C, Yu JX, Zhou A (2003) Dynamically maintaining frequent items over a data stream. In: CIKM
21. Jin R, Agrawal G (2005) An algorithm for in-core frequent itemset mining on streaming data. In: ICDM
22. Karp RM, Shenker S, Papadimitriou CH (2003) A simple algorithm for finding frequent elements in streams and bags. ACM Trans Database Syst 28:51–55
23. Lee D, Lee W (2005) Finding maximal frequent itemsets over online data streams adaptively. In: ICDM
24. Leskovec J, Backstrom L, Kleinberg J (2009) Meme-tracking and the dynamics of the news cycle. In: KDD
25. Lewis DD, Yang Y, Rose TG, Li F (2004) Rcv1: a new benchmark collection for text categorization research. J Mach Learn Res 5:361–397
26. Li H, Lee S, Shan M (2005) Online mining (recently) maximal frequent itemsets over data streams. In: 15th international workshop on research issues in data engineering
27. Li H, Zhang N, Chen Z (2012) A simple but effective maximal frequent itemset mining algorithm over streams. JSW 7(1):25–32
28. Lim Y, Choi J, Kang U (2014) Fast, accurate, and space-efficient tracking of time-weighted frequent items from data streams. In: CIKM
29. Lim Y, Jung M, Kang U (2017) Memory-efficient and accurate sampling for counting local triangles in graph streams: from simple to multigraphs. ACM Trans Knowl Discov Data 11(4)

30. Lim Y, Kang U (2015) Mascot: memory-efficient and accurate sampling for counting local triangles in graph streams. In: KDD
31. Liu H, Lu Y, Han J, He J (2006) Error-adaptive and time-aware maintenance of frequency counts over data streams. In: WAIM
32. Manerikar N, Palpanas T (2009) Frequent items in streaming data: an experimental evaluation of the state-of-the-art. Data Knowl Eng 68(4):415–430
33. Manku GS, Motwani R (2002) Approximate frequency counts over data streams. In: VLDB
34. McAuley JJ, Leskovec J (2013) From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In: WWW
35. Metwally A, Agrawal D, Abbadi AE (2005) Efficient computation of frequent and top-k elements in data streams. In: ICDT
36. Misra J, Gries D (1982) Finding repeated elements. Sci Comput Program 2(2):143–152
37. Vitter JS (1985) Random sampling with a reservoir. ACM Trans Math Softw 11(1):37–57
38. Yamamoto Y, Iwanuma K, Fukuda S (2014) Resource-oriented approximation for frequent itemset mining from bursty data streams. In: SIGMOD
39. Zhang S, Chen L, Tu L (2009a) Frequent items mining on data stream based on time fading factor. In: AICI
40. Zhang S, Chen L, Tu L (2009b) Frequent items mining on data stream using hash-table and heap. In: ICIS



Yongsub Lim received Ph.D. in the School of Computing at KAIST. He worked as a postdoctoral researcher at Data Mining Lab in the Department of Computer Science and Engineering of Seoul National University from September 2015 to June 2016. He is currently working as a researcher at Big Data Tech. Lab of SK Telecom, Republic of Korea. His research interest includes large scale data mining.

U. Kang is an assistant professor in the Department of Computer Science and Engineering of Seoul National University. He received Ph.D. in Computer Science at Carnegie Mellon University, after receiving B.S. in Computer Science and Engineering at Seoul National University. He won 2013 SIGKDD Doctoral Dissertation Award, 2013 New Faculty Award from Microsoft Research Asia, 2016 Korean Young Information Scientist Award, and two best paper awards. He has published over 50 refereed articles in major data mining and database venues. He holds four U.S. patents. His research interests include big data mining.