Efficiently Summarizing Data Streams over Sliding Windows

Nicoló Rivetti, Yann Busnel, Achour Mostefaoui

Efficiently Summarizing Data Streams over Sliding Windows

Nicoló Rivetti
LINA / Université de Nantes,
Nantes, France
Nicolo.Rivetti@univ-nantes.fr

Yann Busnel
Crest (Ensai) / Inria,
Rennes, France
Yann.Busnel@ensai.fr

Achour Mostefaoui
LINA / Université de Nantes,
Nantes, France
Achour.Mostefaoui@univ-nantes.fr

***Abstract*—Estimating the frequency of any piece of information in large-scale distributed data streams became of utmost importance in the last decade (*e.g.*, in the context of network monitoring, big data, *etc.*). If some elegant solutions have been proposed recently, their approximation is computed from the inception of the stream. In a runtime distributed context, one would prefer to gather information only about the recent past. This may be led by the need to save resources or by the fact that recent information is more relevant.**

In this paper, we consider the *sliding window* model and propose two different (on-line) algorithms that approximate the items frequency in the active window. More precisely, we determine a (ε, δ)-additive-approximation meaning that the error is greater than ε only with probability δ. These solutions use a very small amount of memory with respect to the size N of the window and the number n of distinct items of the stream, namely, $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N+\log n))$ and $O(\frac{1}{\tau\varepsilon}\log\frac{1}{\delta}(\log N+\log n))$ bits of space, where τ is a parameter limiting memory usage. We also provide their distributed variant, *i.e.*, considering the *sliding window functional monitoring* model. We compared the proposed algorithms to each other and also to the state of the art through extensive experiments on synthetic traces and real data sets that validate the robustness and accuracy of our algorithms.

I. Introduction and Related Work

In large distributed systems, it is most likely critical to gather various aggregates over data spread across the large number of nodes. This can be modelled by a set of nodes, each observing a stream of items. These nodes have to collaborate to continuously evaluate a given function over the global distributed stream. For instance, current network management tools analyze the input streams of a set of routers to detect malicious sources or to extract user behaviors [1], [2]. The main goal is to evaluate such functions at the lowest cost in terms of the space used at each node, as well as minimizing the update and query time. The solutions proposed so far are focused on computing functions or statistics using ε or (ε, δ)-approximations in poly-logarithmic space over the size m of the stream and the number n of its distinct items.

In the *data streaming* model, many functions have been studied such as the estimation of the number of distinct data items in a stream [3], [4], the frequency moments [5], the most frequent data items [6], the frequency estimation [7], [8] or information divergence over streams [1]. Cormode *et al.* [9] propose solutions for frequency moments estimation in the *functional monitoring* model. In most applications, computing such a function from the inception of a distributed stream is useless [10]. Only the most recent past may be relevant meaning that the function has to be evaluated on part of the stream captured by a window of a given size (say N) that will slide over time. Datar *et al.* [10] introduced the sliding window concept in the data streaming model presenting the *exponential histogram* algorithm that provides an ε-approximation for basic counting. Gibbons and Tirthapura [11] presented an algorithm matching the results of [10] based on the *wave* data structures requiring constant processing time and providing some extensions for distributed streams. Arasu and Manku [12] studied the problem of ε-approximating counts over sliding windows, presenting both deterministic and randomized solutions achieving respectively $O(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon})$ and $O(\frac{1}{\varepsilon}\log\frac{\varepsilon}{\delta})$ space complexity. In this model there are also works on variance [13], quantiles [12] and frequent items [14]. Merging both models, [15] provides an optimal solution for the heavy hitters problem in the sliding window functional monitoring model.

In this paper, we tackle the frequency estimation problem in the sliding window model. Whatever is the model, this problem cannot be reduced to the heavy hitters (frequent items) problem and approximate counts. Indeed, having the frequency estimation of items allows to determine frequent element but the converse does not hold. Moreover, using little memory (low space complexity) implies some kind of data aggregation. If the number of counters is less than the number of different items then necessarily each counter encodes the occurrences of more than one item. The problem is then how to slide the window to no more keep track of the items that exited the window and how to introduce new items. As a consequence, our work cannot be compared to [14], [16]. To our knowledge the only work that tackles a similar problem is [17]. Their proposal, named ECM-sketches, consists in a compact structure combining some state-of-the-art sketching techniques for data stream summarization, with sliding window synopses.

We extend the well-known algorithm for frequency estimation, namely the Count-Min sketch [8], in a windowed version. We propose our approach in two steps, two first naive and straightforward algorithms called Perfect and Simple followed by two more sophisticated ones called Proportional windowed and Splitter windowed algorithms. Then, we compare their respective performances together with the ECM-sketches solution, proposed in [17].

This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003), and by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01).

This paper is composed of 5 Sections. Section II describes the computational model and some necessary background. In Section III, after two naive first step algorithms, we propose two novel (ε, δ)-additive-approximations, achieving respectively $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N + \log n))$ and $O(\frac{1}{\tau\varepsilon}\log\frac{1}{\delta}(\log N + \log n))$ bits[1] of space, where τ is an additional parameter limiting memory usage (see Section III-D). Section III-E and Section III-F present respectively the distributed variant and the time-based sliding windows extention. The efficiency of the three algorithms and the algorithm proposed in [17] are analyzed and Section IV presents an extended performance evaluation of the estimation accuracy of our algorithms, with both synthetic traces and real data sets, inspired by [18].

II. Preliminaries and Background

A. Data Streaming Model

We present the computation model under which we analyze our algorithms and derive bounds: the data streaming model [19]. We consider a massively long input stream σ, that is, a sequence of elements $\langle a_1, a_2, \ldots, a_m, \ldots \rangle$ called samples. Samples are drawn from a universe $[n] = \{1, 2, \ldots, n\}$ of items. The size of the universe (or number of distinct items) of the stream is n. This sequence can only be accessed in its given order (no random access). The problem to solve can be seen as a function ϕ evaluated on a sequence of items prefix of size m of a stream σ under memory constraints. For example if the function ϕ represents the most frequent item then the function ϕ applied to the first m items of the stream returns the most frequent item among these m first samples.

In order to reach these goals, we rely on randomized algorithms that implement approximations of the desired function ϕ. Namely, such an algorithm $\mathcal{A}$ evaluates the stream in a single pass (on-line) and continuously. It is said to be an (ε, δ)-additive-approximation of the function ϕ on a stream σ if, for any prefix of size m of items of the input stream σ, the output $\hat{\phi}$ of $\mathcal{A}$ is such that $\mathbb{P}\{\mid \hat{\phi} - \phi \mid > \varepsilon C\} < \delta$, where ε, $\delta > 0$ are given as precision parameters and C is an arbitrary constant. The parameter ε represents the precision of the estimation of the approximation. For instance $\varepsilon = 0.1$ means that the additive error is less than 10% and $\delta = 0.01$ means that this approximation will not be satisfied with a probability less than 1%.

On the other hand, as explained in the Introduction, we are only interested in the recent past. This is expressed by the fact that when the function ϕ is evaluated, it will be only on the N more recent items among the m items already observed, that is, the *sliding window* model formalized by Datar *et al.* [10]. In this model, samples arrive continuously and expire after exactly N steps. A step corresponds to a sample arrival, *i.e.*, we consider count-based sliding windows. The challenge consists in achieving this computation in sub-linear space. When N is set to the maximal value of m, the sliding window model boils down to the classical model. The supplemental problem brought by a sliding window resides in the fact that when a prefix of a stream is summarized we lose the temporal information related to the different items making the exclusion of the most ancient items non trivial with little memory.

[1]For the sake of clarity, we will use the notation log to denote the logarithm in base 2 for the rest of this paper.

B. Vanilla Count-Min Sketch

The problem we tackle in this paper is the *frequency estimation* problem. In a stream, each item appears a given number of times that allows to define its frequency. The function that defines this problem returns a frequency vector $\mathbf{f} = (f_1, \ldots, f_n)$ where f_j represents the number of occurrences of item j in the portion of the input stream σ evaluated so far. The goal is to provide an estimate $\hat{f}_j$ of f_j for each item $j \in [n]$.

Cormode and Muthukrishnan have introduced in [8] the COUNT-MIN sketch that provides, for each item j an (ε, δ)-additive-approximation $\hat{f}_j$ of the frequency f_j. This algorithm leverages collections of 2-universal hash functions. Recall that a collection $\mathbb{H}$ of hash functions $h : [M] \to [M']$ is said to be 2-universal if for every 2 distinct items $x, y \in [M]$, $\mathbb{P}_{h \in \mathbb{H}}\{h(x) = h(y)\} \leq \frac{1}{M'}$, that is, the collision probability is as if the hash function assigns truly random values to any $x \in [M]$. Carter and Wegman [20] provide an efficient method to build large families of hash functions approximating the 2-universality property.

The VANILLA COUNT-MIN sketch consists of a two dimensional *count* matrix of size $c_1 \times c_2$, where $c_1 = \lceil \log \frac{1}{\delta} \rceil$ and $c_2 = \lceil \frac{e}{\varepsilon} \rceil$. Each row is associated with a different 2-universal hash function $h_i : [n] \to [c_2]$. When it reads sample j, it updates each row: $\forall i \in [c_1], count[i, h_i(j)] \leftarrow count[i, h_i(j)] + 1$. That is, the cell value is the sum of the frequencies of all the items mapped to that cell. Since each row has a different collision pattern, upon request of $\hat{f}_{j'}$ we want to return the cell associated with j' minimising the collisions impact. In other words, the algorithm returns, as $f_{j'}$ estimation, the cell associated with j' with the lowest value: $\hat{f}_{j'} = \min_{1 \leq i \leq c_1}\{count[i, h_i(j')]\}$.

Fed with a stream of m items, the space complexity of this algorithm is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m + \log n))$ bits, while update and query time complexities are $O(\log 1/\delta)$. Concerning its accuracy, the following bound holds: $\mathbb{P}\{\mid \hat{f}_j - f_j \mid \geq \varepsilon(m - f_j)\} \leq \delta$, while $f_j \leq \hat{f}_j$ is always true.

III. Windowed Count-Min

The COUNT-MIN algorithm solves brilliantly the frequency estimation problem. We propose two extensions in order to meet the sliding window model: PROPORTIONAL and SPLITTER. Nevertheless, we first introduce two naive algorithms that enjoy optimal bounds with respect to accuracy (algorithm PERFECT) and space complexity (algorithm SIMPLE). Note that in the following f_j is redefined as the frequency of item j in the last N samples among the m items of the portion of the stream evaluated so far.

Due to space constraints, some algorithm pseudo-codes and some proofs are available in the companion paper [21], which the interested reader is invited to consult.

A. Perfect Windowed Count-Min

PERFECT provides the best accuracy by dropping the complexity space requirements: it trivially stores the whole active window in a queue. When it reads sample j, it enqueues j and increases all the *count* matrix cells associated with j. Once the queue reaches size N, it dequeues the expired sample j' and decreases all the cells associated with j'. The frequency estimation is retrieved as in the VANILLA COUNT-MIN (*cf.* Section II-B).

Theorem 3.1: PERFECT is an (ε, δ)-additive-approximation of the frequency estimation problem in the count-based sliding window model where $\mathbb{P}\{\mid \hat{f}_j - f_j \mid \geq \varepsilon(N - f_j)\} \leq \delta$, while $f_j \leq \hat{f}_j$ is always true.

Theorem 3.2: PERFECT space complexity is $O(N)$ bits, while update and query time complexities are $O(\log 1/\delta)$.

B. Simple Windowed Count-Min

SIMPLE is as straightforward as possible and achieves optimal space complexity with respect to the vanilla algorithm. It behaves as the VANILLA COUNT-MIN, except that it resets the $count$ matrix at the beginning of each new window. Obviously it provides a really rough estimation since it simply drops all information about any previous window once a new window starts.

Theorem 3.3: SIMPLE space complexity is $O(\frac{1}{\varepsilon} \log \frac{1}{\delta} (\log N + \log n))$ bits, while update and query time complexities are $O(\log 1/\delta)$.

C. Proportional Windowed Count-Min

We now present the first extension algorithm, denoted PROPORTIONAL. The intuition behind this algorithm is as follows. At the end of each window, it stores separately a snapshot of the $count$ matrix, which represents what happened during the previous window. Starting from the current $count$ state, for each new sample, it increments the associated cells and decreases all the $count$ matrix cells proportionally to the last snapshot. This smooths the impact of resetting the $count$ matrix throughout the current window.

More formally, after reading N samples, PROPORTIONAL stores the current $count$ matrix and divides each cell by the window size: $\forall i_1, i_2 \in [c_1] \times [c_2],\ snapshot[i_1, i_2] \leftarrow count[i_1, i_2]/N$. This snapshot represents the average step increment of the $count$ matrix during the previous window. When PROPORTIONAL reads sample j, it increments the $count$ cells associated with j as in the VANILLA COUNT-MIN and subtracts $snapshot$ from $count$: $\forall i_1, i_2 \in [c_1] \times [c_2],\ count[i_1, i_2] \leftarrow count[i_1, i_2] - snapshot[i_1, i_2]$. Finally, the frequency estimation is retrieved from $count$ as in the vanilla algorithm.

Theorem 3.4: PROPORTIONAL space complexity is $O(\frac{1}{\varepsilon} \log \frac{1}{\delta}(\log N + \log n))$ bits. Update and query time complexities are $O(\frac{1}{\varepsilon} \log 1/\delta)$ and $O(\log 1/\delta)$.

D. Splitter Windowed Count-Min

PROPORTIONAL removes the *average* frequency distribution of the previous window from the current window. Consequently, PROPORTIONAL does not capture sudden changes in the stream distribution. To cope with this flaw, one could track these critical changes through multiple snapshots. However, each row of the $count$ matrix is associated with a specific 2-universal hash function, thus changes in the stream distribution will not affect equally each rows.

Therefore, SPLITTER proposes a finer grained approach analyzing the update rate of each cell in the $count$ matrix. To

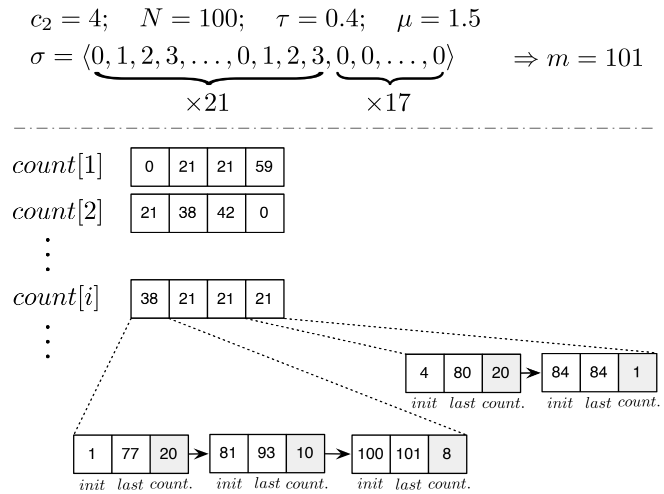


Fig. 1: State of the data structure of SPLITTER after a prefix of 101 items of σ.

record changes in the cell update rate, we add a (fifo) queue of sub-cells to each cell. When SPLITTER detects a relevant variation in the cell update rate, it creates and enqueues a new sub-cell. This new sub-cell then tracks the current update rate, while the former one stores the previous rate.

Each sub-cell has a frequency $counter$ and 2 timestamps: $init$, that stores the (logical) time where the sub-cell started to be active, and $last$, that tracks the time of the last update. After a short bootstrap, any cell contains at least two sub-cells: the current one that depicts what happened in the very recent history, and a predecessor representing what happened in the past. Listing III.1 presents the global behavior of SPLITTER, while Figure 1 illustrates a possible state of the data structure of SPLITTER, after reading a prefix of 101 items of σ, which is introduced in the top part of the figure with all the parameters of SPLITTER.

SPLITTER spawns additional sub-cells to capture distribution changes. The decision whether to create a new sub-cell is tuned by two parameters, τ and μ, and an error function: ERROR. Informally, the function ERROR evaluates the potential amount of information lost by merging two consecutive sub-cells, while μ represents the amount of affordable information loss. Performing this check at each sample arrival may lead to erratic behaviors. To avoid this, we introduced τ, such that $0 < \tau \leq 1$, that sets the minimal length ratio of a sub-cell before taking this sub-cell into account in the decision process.

In more details, when SPLITTER reads sample j, it has to phase out the expired data from each sub cell. Then, for each cell of $count$, it retrieves the oldest sub-cell in the queue, denoted $first$ (Line 9). If $first$ was active precisely N steps ago (Line 10), then it computes the rate at which $first$ has been incremented while it was active (Line 11). This value is subtracted from the cell counter v (Line 12) and from $first$ counter (Line 13). Having retracted what happened N steps ago, $first$ moves forward increasing its $init$ timestamp (Line 14). Finally, $first$ is removed if it has expired (Lines 15 and 16).

The next part handles the update of the cells associated with item j. For each of them (Line 19), SPLITTER increases

Listing III.1: SPLITTER WINDOWED COUNT-MIN

1: **init do**
2: $\quad count[1\dots c_1][1\dots c_2] \leftarrow \overrightarrow{\langle\emptyset, 0\rangle}$ ▷ the set is a queue
3: $\quad$ Choose c_1 independent hash functions $h_1 \dots h_{c_1} : [n] \to [c_2]$ from a 2-universal family.
4: $\quad m' \leftarrow 0$
5: **end init**
6: **upon** $\langle Sample \mid j\rangle$ **do**
7: $\quad$ **for** $i_1 = 1$ to c_1 and $i_2 = 1$ to c_2 **do**
8: $\quad\quad \langle queue, v\rangle \leftarrow count[i_1, i_2]$
9: $\quad\quad first \leftarrow$ head of $queue$
10: $\quad\quad$ **if** $\exists first \wedge first_{init} = m' - N$ **then**
11: $\quad\quad\quad v' \leftarrow \frac{first_{counter}}{first_{last} - first_{init} + 1}$
12: $\quad\quad\quad v \leftarrow v - v'$
13: $\quad\quad\quad first_{counter} \leftarrow first_{counter} - v'$
14: $\quad\quad\quad first_{init} \leftarrow first_{init} + 1$
15: $\quad\quad\quad$ **if** $first_{init} > first_{last}$ **then**
16: $\quad\quad\quad\quad$ removes $first$ from queue
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end if**
19: $\quad\quad$ **if** $h_{i_1}(j) = i_2$ **then**
20: $\quad\quad\quad v \leftarrow v + 1$
21: $\quad\quad\quad last \leftarrow$ bottom of $queue$
22: $\quad\quad\quad$ **if** $\nexists last$ **then**
23: $\quad\quad\quad\quad$ Creates and enqueues a new sub-cell
24: $\quad\quad\quad$ **else if** $last_{counter} < \frac{\tau N}{c_2}$ **then**
25: $\quad\quad\quad\quad$ Updates sub-cell $last$
26: $\quad\quad\quad$ **else**
27: $\quad\quad\quad\quad pred \leftarrow$ predecessor of $last$ in $queue$
28: $\quad\quad\quad\quad$ **if** $\exists pred \wedge \text{ERROR}(pred, last) \le \mu$ **then**
29: $\quad\quad\quad\quad\quad$ Merges $last$ into $pred$ and renews $last$
30: $\quad\quad\quad\quad$ **else**
31: $\quad\quad\quad\quad\quad$ Creates and enqueues a new sub-cell
32: $\quad\quad\quad\quad$ **end if**
33: $\quad\quad\quad$ **end if**
34: $\quad\quad$ **end if**
35: $\quad\quad count[i_1, i_2] \leftarrow \langle queue, v\rangle$
36: $\quad$ **end for**
37: $\quad m' \leftarrow m' + 1$
38: **end upon**
39: **function** GETFREQ(j) ▷ returns $\hat{f}_j$
40: $\quad$ **return** round$\{\min\{count[i][h_i(j)].v \mid 1 \le i \le c_1\}\}$
41: **end function**

the cell counter v (Line 20) and retrieves the current sub-cell, denoted $last$ (Line 21). **(a)** If $last$ does not exist, it creates and enqueues a new sub-cell (Line 23). **(b)** If $last$ has not reached the minimal size to be evaluated (Line 24), $last$ is updated (Line 25). **(c)** If not, SPLITTER retrieves the predecessor of $last$: $pred$ (Line 27). **(c.i)** If $pred$ exists and the amount of information lost by merging is lower than the threshold μ (Line 28), SPLITTER merges $last$ into $pred$ and renews $last$ (Line 29). **(c.ii)** Otherwise it creates and enqueues a new sub-cell (Line 31), *i.e.*, it *splits* the cell.

Lemma 3.5: [Number of Splits Upper-bound] Given $0 < \tau \le 1$, the maximum number $\overline{s}$ of splits (number of sub-cells spawned to track distribution changes) is $O(\frac{1}{\varepsilon\tau}\log\frac{1}{\delta})$.

Proof: A sub-cell is not involved in the decision process of merging or splitting while its counter is lower than $\frac{\tau N}{c_2} = \varepsilon\tau N$. So, no row can own more than $\frac{1}{\varepsilon\tau}$ splits. Thus, the maximum numbers of splits among the whole data structure $count$ is $s = O(\frac{1}{\varepsilon\tau}\log\frac{1}{\delta})$. ■

Theorem 3.6: SPLITTER space complexity is $O(\frac{1}{\tau\varepsilon}\log\frac{1}{\delta}(\log N + \log n))$ bits, while update and query time complexities are $O(\log 1/\delta)$.

Proof: Each cell of the $count$ matrix is composed of a counter and a queue of sub-cells made of two timestamps and a counter, all of size $O(\log N)$ bits[2]. Without any split and considering that all cells have bootstrapped, the initial space complexity is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N + \log n))$ bits. Each split costs two timestamps and a counter (size of a sub-cell). Let s be the number of splits, we have $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N + \log n) + s\log N)$ bits. Lemma 3.5 establishes the following space complexity bound: $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N + \log n) + \frac{1}{\varepsilon\tau}\log\frac{1}{\delta}\log N)$ bits.

Each update requires to access each of the $count$ matrix cells in order to move the sliding window forward. However, we can achieve the same result by performing this phase-out operation (from Line 10 to Line 18) only on the $count$ matrix cells that are accessed by the update and query procedures. Given this optimization, update and query require to lookup one cell by row of the $count$ matrix. Then, the query and update time complexities are $O(\log 1/\delta)$. ■

One can argue that sub-cell creations and destructions cause memory allocations and disposals. However, we believe that it is possible to avoid wild memory usage leveraging the sub-cell creation patterns, either through a smart memory allocator or a memory aware data structure.

Finally, Table I summarizes the space, update and query complexities of the presented algorithms.

E. DISTRIBUTED COUNT-MIN

The *functional monitoring* model [9] extends the data streaming model by considering a set of k nodes, each receiving an inbound stream σ_ℓ ($\ell \in [k]$). These nodes interact only with a specific node called *coordinator*.

Notice that the $count$ matrix is a linear-sketch data structure, which means that for every two streams σ_1 and σ_2, we have COUNT-MIN$(\sigma_1 \cup \sigma_2) =$ COUNT-MIN$(\sigma_1) \oplus$ COUNT-MIN(σ_2), where $\sigma_1 \cup \sigma_2$ is a stream containing all the samples of σ_1 and σ_2 in any order, and $\oplus$ sums the underlying $count$ matrix term by term. Considering only the last N samples of σ_1 and σ_2, the presented algorithms are also linear-sketches.

The sketch property is suitable for the distributed context. Each node can run locally the algorithm on its own stream σ_ℓ ($\ell \in [k]$). The coordinator can retrieve all the $count_\ell$ matrices ($\ell \in [k]$), sum them up and obtain the global matrix $\overline{count} = \bigoplus_{\ell\in[k]} count_\ell$. The coordinator is then able to retrieve the frequency estimation for each item on the global distributed stream $\overline{\sigma} = \sigma_1 \cup \ldots \cup \sigma_k$.

Taking inspiration from [15], we can define the DISTRIBUTED COUNT-MIN (DCM) algorithm, which sends the $count$ matrix to the coordinator each εN samples. DCM can be applied to the four aforementioned windowed extensions of VANILLA COUNT-MIN, resulting in a distributed frequency

[2]Note that, for the sake of clarity, timestamps are of size $O(\log m)$ bits in the pseudo-code while counters of size $O(\log N)$ bits are sufficient.

TABLE I: Complexities comparison

Algorithm	Space (bits)	Update time	Query time
VANILLA COUNT-MIN [8]	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n))$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
PERFECT	$O(N)$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
SIMPLE	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N+\log n))$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
PROPORTIONAL	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N+\log n))$	$O(\frac{1}{\varepsilon}\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
SPLITTER	$O(\frac{1}{\tau\varepsilon}\log\frac{1}{\delta}(\log N+\log n))$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
ECM-SKETCH [17]	$O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\left(\log^2\varepsilon N+\log n\right)\right)$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$

(ε,δ)-additive-approximation in the *sliding windowed functional monitoring* model.

Theorem 3.7: DCM communication complexity is $O(\frac{k}{\varepsilon^2}\log\frac{1}{\delta}\log N)$ bits per window.

Theorem 3.8: DCM introduces an additive error of at most $k\varepsilon N$, *i.e,* the skew between any cell (i_1,i_2) of the global $\overline{count}$ matrix at the coordinator and the sum of the cells (i_1,i_2) of the $count_\ell$ matrices $(\ell\in[k])$ on nodes is at most $k\varepsilon N$.

F. Time-based windows

We have presented the algorithms assuming count-based sliding windows, however all of them can be easily applied to time-based sliding windows. Recall that in time-based sliding windows the steps defining the size of the window are time ticks instead of sample arrivals.

In each algorithm it is possible to split the update code into the subroutine increasing the $count$ matrix and the subroutine phasing out expired data (*i.e.*, decreasing the $count$ matrix). Let denote the former as UPDATESAMPLE and the latter as UPDATETICK. At each sample arrival, the algorithm will perform the UPDATESAMPLE subroutine, while performing the UPDATETICK subroutine at each time tick. Note that timestamps have to be updated using the current time tick count.

This modification affects the complexities of the algorithms, since N is no longer the number of samples, but the number of time ticks. Thus, the complexities improve or worsen, depending if the number of sample arrivals per time tick is greater or lower than 1.

IV. PERFORMANCE EVALUATION

This section provides the performance evaluation of our algorithms. We have conducted a series of experiments on different types of streams and parameter settings. To verify the robustness of our algorithms, we have fed them with synthetic traces and real-world datasets. The latter give a representation of some existing monitoring applications, while synthetic traces allow to capture phenomena that may be difficult to obtain otherwise. Each run has been executed a hundred times, and we provide the mean over the repeated runs, after removing the 1st and 10th deciles to avoid outliers.

A. Settings

If not specified otherwise, in all experiments, the window size is $N=50,000$ and streams are of length $m=3N$ (*i.e.* $m=150,000$) with $n=1,000$ distinct items. Note that we restrict the stream to 3 windows since the behavior of the algorithms in the following windows does not change, as each algorithm relies only on the latest past window. We skip the first window where all algorithms are trivially perfect.

The VANILLA COUNT-MIN uses two parameters: δ that sets the number of rows c_1, and ε, which tunes the number of columns c_2. In all simulations, we have set $\varepsilon=0.1$, meaning $c_2=\lceil\frac{e}{0.1}\rceil=28$ columns. Most of the time, the $count$ matrix has several rows. However, analyzing results using multiple rows requires taking into account the interaction between the hash functions. If not specified, for the sake of clarity, we present the results for a single row ($\delta=0.5$).

In order to simulate changes in the distribution over time, our stream generator considers a period p, a width w and a number of shifts r as parameters. After every p samples, the distribution is shifted right (from lower to greater items) by w positions. Then, after r shifts, the distribution is reset to the initial unshifted version. If not specified, the default settings are $w=2c_1$, $p=10,000$ and $r=4$.

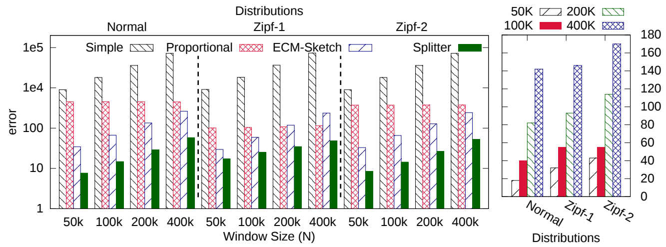
We evaluate the performance by generating families of synthetic streams, following four distributions: **(i) Uniform**: uniform distribution; **(ii) Normal**: truncated standard normal distribution; **(iii) Zipf-1**: Zipfian distribution with $\alpha=1.0$; and **(iv) Zipf-2**: Zipfian distribution with $\alpha=2.0$.

We compare SPLITTER with the other presented algorithm, namely PERFECT SPLITTER and PROPORTIONAL, as well as with the ECM-SKETCH algorithm proposed by Papapetrou *et al.* [17].

The wave-based [11] version of ECM-SKETCH that we have implemented replaces each counter of the $count$ matrix with a *wave* data structure. Each wave is a set of lists, the number and the size of such lists is set by the parameter ε_{wave}. Then, setting $\varepsilon_{wave}=\varepsilon$, the wave-based ECM-SKETCH space complexity is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}\left(\frac{1}{\varepsilon}\log^2\varepsilon N+\log n\right))$ bits.

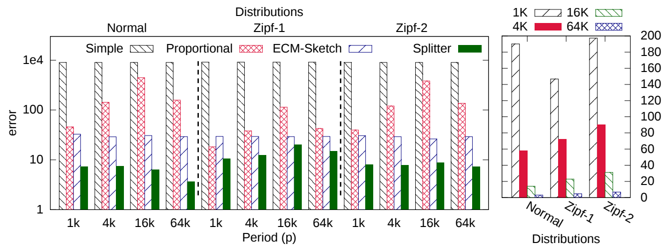
Moreover, recall that SPLITTER has two additional parameters: μ and τ. We provide the results for $\mu=1.5$ and $\tau=0.05$. Their influence is analyzed separately in Section IV-C. Given these parameters, we have an upper bound of at most $\bar{s}=560$ spawned sub-cells (*cf.* Lemma 3.5). With the parameters stated so far and the provided memory usage upper bounds, ECM-SKETCH uses at least twice the memory required by SPLITTER. Notice however that the upper bound of $\bar{s}=560$ spawned sub-cells is never reached in any test. Acording to our experiments, ECM-SKETCH uses at least 4.5 times the memory required by SPLITTER in this evaluation.

Finally, the accuracy metric used in our evaluation is the mean absolute error of the frequency estimation of all n items



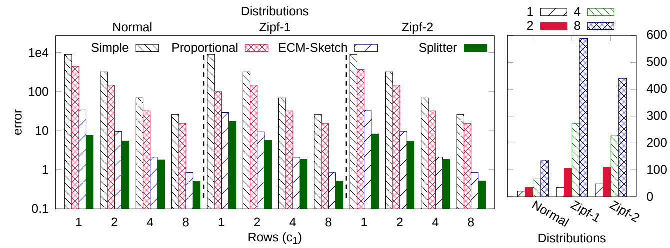
(a) Average estimation error (b) Average splits

Fig. 2: Results for different window sizes (N)



(a) Average estimation error (b) Average splits

Fig. 3: Results for different periods (p)



(a) Average estimation error (b) Average splits

Fig. 4: Results for different number of rows (c_1)

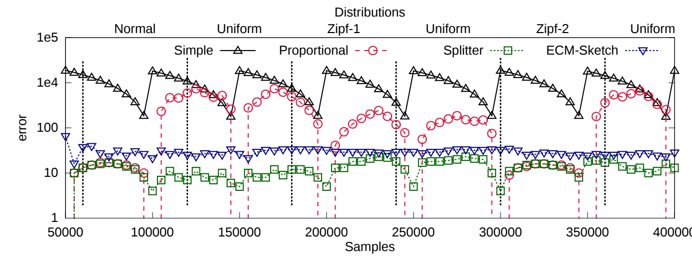


Fig. 5: Estimation error with multiple distributions

returned by the algorithms with respect to PERFECT, that is $\left(\sum_{j\in[n]}\left|\hat{f}_j^{\text{PERFECT}} - \hat{f}_j^{\text{TESTEDALGORITHM}}\right|\right)/n$. We refer to this metric as *estimation error*. We also evaluate the additional space used by SPLITTER, due to the merge and split mechanisms, through the exact number of splits s.

B. Performance comparison

a) Window sizes: Figure 2(a) presents the estimation error of the SIMPLE, PROPORTIONAL, SPLITTER and ECM-SKETCH algorithms considering the Normal, Zipf-1 and Zipf-2 distributions, with $N = 50,000$ (and *a fortiori* $m = 150,000$), $N = 100,000$ (with $m = 300,000$), $N = 200,000$ (with $m = 600,000$) and $N = 400,000$ (with $m = 1,200,000$). Note that the y-axis (*error*) is in logarithmic scale and error values are averaged over the whole stream. SIMPLE is always the worst (with an error equals to 3395 in average), followed by PROPORTIONAL (451 in average), ECM-SKETCH (262 in average) and SPLITTER (57 in average). In average, SPLITTER error is 4 times smaller than ECM-SKETCH, with 4 times less memory requirement. The error estimation of SIMPLE, PROPORTIONAL, ECM-SKETCH and SPLITTER increases in average respectively with a factor 2.0, 1.1, 1.9 and 1.7 for each 2-fold increase of N.

Figure 2(b) gives the number of splits spawned by SPLITTER in average to keep up with the distribution changes. The number of splits grows in average with a factor 1.7 for each each 2-fold increase of N. In fact, as τ is fixed, the minimal size of each sub-cell grows with N, and so does the error.

b) Periods: Recall that the distribution is shifted each p samples. The estimation error and the number of splits for $p \in \{1,000; 4,000; 16,000; 64,000\}$ are displayed in Figure 3. Again, SPLITTER (20 at most) is always better than ECM-SKETCH (26 at best) achieving roughly a 4 fold improvement. SIMPLE is always the worst (more than 900), followed by PROPORTIONAL (roughly 140 in average). In more details, PROPORTIONAL grows from $1,000$ to $16,000$, because slower shifts cast the error on less items, resulting in a larger mean absolute error. However, for $64,000$ we have less than a shift per window, meaning that some window will have a non-changing distribution and PROPORTIONAL will be almost perfect. In general SPLITTER estimation error is not heavily affected by decreasing p since it keeps up by spawning more sub-cells. For $p = 64,000$ we have at most 7 splits, while for $p = 1,000$ we have in average 166 splits. Each 4-fold decrease of p increases the number of splits by $3.4\times$ in average.

c) Rows: The COUNT-MIN algorithm uses a hash-function for each row mapping items to cells. Using multiple rows produces different collisions patterns, increasing the accuracy. Figure 4 presents the estimation error and splits for $c_1 = 1$ (meaning that $\delta = 0.5$), $c_1 = 2$ ($\delta = 0.25$), $c_1 = 4$ ($\delta = 0.0625$) and $c_1 = 8$ rows ($\delta = 0.004$). Increasing the number of rows do enhance the accuracy of the algorithms. However, the ordering among the algorithms does not change: SIMPLE, PROPORTIONAL, ECM-SKETCH and SPLITTER achieve respectively 331, 126, 11 and 4 in average. For each distribution shift, $2w$ items change their occurrence probability, meaning that (without collisions) most likely $2wc_1$ cells will change their update rate. Since $w = 2c_1$, we have $4c_1^2$ potential splits per shift. Hopefully, experiments illustrate that the number of splits growth is not quadratic: in average it increases by $2.4\times$ for each 4-fold increase of c_1.

d) Multiple distributions: This test on a synthetic trace has $p = 15,000$ and swaps the distribution each $60,000$ samples in the following order: Uniform, Normal, Uniform, Zipf-1, Uniform, Zipf-2, Uniform. The streams is of length $m = 400,000$. Note that, in order to avoid side effect, the distribution shift and swap periods are not synchronised with the window size ($N = 50,000$).

Figure 5 presents the estimation error evolution as the stream unfolds. SPLITTER error does not exceed 23 (and is

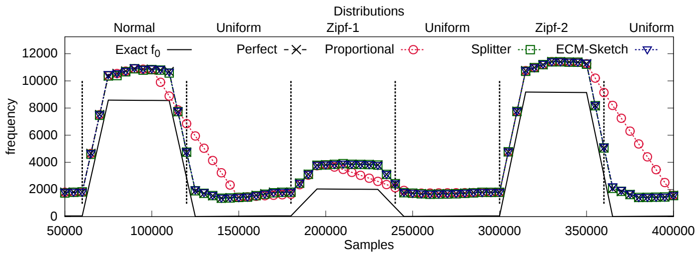


Fig. 6: Estimation of item 0 with multiple distributions

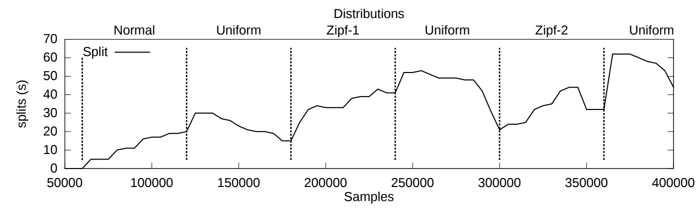


Fig. 7: Number splits s with multiple distributions

equal to 13 in average). ECM-SKETCH maximum error is 65 (29 in average), as PROPORTIONAL goes up to 740 (207 in average) and SIMPLE reaches 1877 (1035 in average). Since at the beginning of each window SIMPLE resets its *count* matrix, there is a periodic behavior: the error burst when a window starts and shrinks towards the end. In the 1-st window period (0 to $50,000$) and in the 6-th windows ($250,000$ to $300,000$) the distribution does not change over time (shifting Uniform has no effect). This means that SPLITTER does not capture more information than PROPORTIONAL, thus they provide the same estimations in the 2-nd and the 7-th windows (respectively between $50,000$ and $100,000$ samples then between $300,000$ and $350,000$ samples).

Figure 6 presents the value of f_0 and its estimations over time (for clarity SIMPLE is omitted). The plain line represents the exact value of f_0 according to time, which also reflects the distribution changes. The plots for PERFECT, ECM-SKETCH and SPLITTER are overlapping (exes, nablas and squares). Except for the error introduced by the COUNT-MIN approximation, they all follow the f_0 shape precisely. However, even that is not clearly visible on Figure 6, notice that ECM-SKETCH error is always larger than that of SPLITTER. More precisely, one should observe that item 0 probability of occurrence changes significantly in the following intervals: $[60k, 75k]$, $[180k, 195k]$ and $[300k, 315k]$. PROPORTIONAL fails to follow the f_0 trend in the windows following those intervals, namely the 3-rd, 5-th and 8-th, since it is unable to correctly assess the previous window distribution.

Finally, Figure 7 presents the number of splits s according to time. There are in average 51 and at most 73 splits (while the theoretical upper bound $\overline{s}$ is 560 according to Lemma 3.5). Interestingly enough, splits decrease when the distribution does not change (in the Uniform intervals for instance). That means that, as expected, some sub-cells expire and no new sub-cells are created. In other words, SPLITTER correctly detects that no changes occur. Conversely, when a distribution shifts or swaps, there is a steep growth, *i.e*, the change is detected. This pattern is clearly visible in the 2-nd window.

e) DDoS: As illustrated in the Global Iceberg problem [18], tracking most frequent items in distributed data streams is not sufficient to detect Distributed Denial of Service (DDoS). As such, one should be able to estimate the frequency of any item. To evaluate our algorithm in this use-case, we have retrieved the CAIDA "DDoS Attack 2007" [22] and "Anonymized Internet Traces 2008" [23] datasets, interleaved them and retained the first $400,000$ samples (*i.e.*, the DDoS attack beginning). The stream is composed by $n = 4.9 \times 10^4$ distinct items. The item representing the DDoS target has a frequency proportion equal to 0.09, while the second most frequent item owns a 0.004 frequency proportion. Figure 8 presents the estimation error evolution over time. In order to avoid drowning the estimation error in the high number of items, we have restricted the computation to the most frequent 7500 items, which cover 75% of the stream[3]. Figure 8 illustrates some trends similar to the previous test, however the estimation provided by PROPORTIONAL, ECM-SKETCH and SPLITTER are quite close since the stream changes much less over time. SIMPLE does not make less error than 178 (that is 1002 in average), while PROPORTIONAL, ECM-SKETCH and SPLITTER do not exceed respectively 73 (34 in average), 53 (33 in average) and 25 (16 in average). On the other hand, for SPLITTER, there are at most 154 splits with an average of 105 splits.

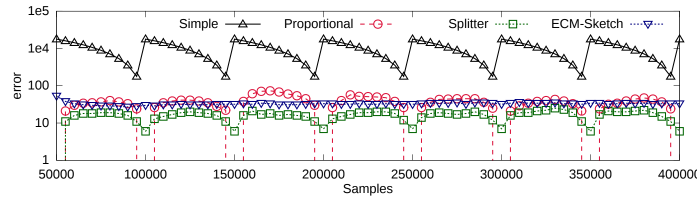
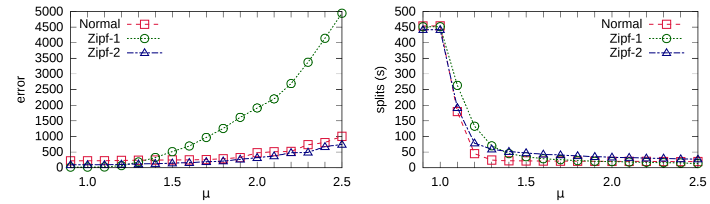


Fig. 8: Results for the DDoS trace



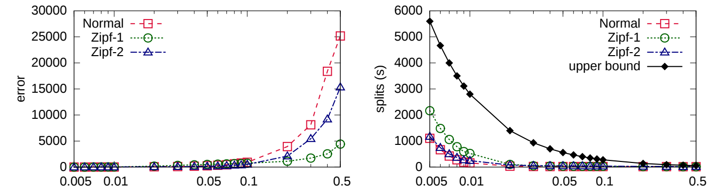
(a) Average estimation error (b) Average number of splits

Fig. 9: Performance comparison with $\tau = 0.05$.

C. Impact of the Splitter parameters

Figure 9 presents the estimation error and the number of splits with several values of $\mu \in \{0.9, 2.5\}$ and a fixed $\tau = 0.05$. As expected, the estimation error grows with μ. Zipf-1

[3]The remaining items have a frequency proportion lower than 2×10^{-5}.



(a) Average estimation error (b) Average number of splits

Fig. 10: Performance comparison with $\mu = 1.5$.

goes from 18 ($\mu = 0.9$) to $4,944$ ($\mu = 2.5$), while the other distributions in average go from 110 ($\mu = 0.9$) to 684 ($\mu = 2.5$). Conversely, increasing μ decreases the number of splits. Since ERROR cannot return a value lower than 1.0, going from 1.0 to 0.9 has almost no effect with at most 454 splits, which represents roughly 19% less than the theoretical upper bound. From $\mu = 1.0$ to 1.3, the average falls down to 51, reaching 20 at $\mu = 2.5$. There is an obvious tradeoff around $\mu = 1.5$ that should represents a nice parameter choice for a given user.

Figure 10 presents the estimation error and the number of splits according to the parameter $\tau \in \{0.005, 0.5\}$, with a fixed $\mu = 1.5$. Note that the x-axis (τ) is logarithmic. As for μ, the estimation error increases with τ: the average starts at 4 (with $\tau = 0.005$), reaches 610 at $\tau = 0.1$ and grows up at $12,198$ (for $\tau = 0.5$). Conversely, increasing τ decreases the number of splits: the average starts at $1,659$ ($\tau = 0.005$), reaches 77 at $\tau = 0.02$ and ends up at 14 ($\tau = 0.5$). In order to illustrate the accuracy of our splitting heuristic, Figure 10(b) shows also the theoretical upper bound. Again, there seems to be a nice tradeoff around $\tau = 0.05$, letting a user having his cake and eat it too!

To summarize, the trend in all the last four plots (and the results for different values of p and c_1) hints to the existence of some optimal value of μ and τ that should minimise the error and the splits. This optimal value seems to either be independent from the stream distribution or computed based on the recent behavior of the algorithm and some constraints provided by the user. Seeking for a extensive analysis of this optimum represents a challenging open question.

V. CONCLUSION AND FUTURE WORK

We have presented two (ε, δ)-additive-approximations for the frequency estimation problem in the sliding windowed data streaming model: PROPORTIONAL and SPLITTER. They have a space complexity of respectively $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log N + \log n))$ and $O(\frac{1}{\tau\varepsilon}\log\frac{1}{\delta}(\log N + \log n))$ bits , while their update and query time complexities are $O(\log\frac{1}{\delta})$.

Leveraging the *sketch* property, we have shown how to apply our proposal to distributed data streams, with a communication cost of $O(\frac{k}{\varepsilon^2}\log\frac{1}{\delta}\log N)$ bits per window. However, we believe that there is still room for improvement.

We have performed an extensive performance evaluation to compare their respective efficiency and also to compare them to the only similar work in the related works. This study shows the accuracy of both algorithms and that they outperform the only existing solution with real world traces and also with specifically tailored adversarial synthetic traces. Last but not least, these results reach better estimation with respect to the state of the art proposal and required 4 times less memory usage. We have also studied the impact of the two additional parameters of the SPLITTER algorithm (τ and μ).

From these results, we are looking forward an extensive formal analysis of the approximation and space bounds of our algorithms. In particular, we seek some insight for computing the optimal values of τ and μ, minimizing the space usage and maximizing the accuracy of SPLITTER.

REFERENCES

[1] E. Anceaume and Y. Busnel, "A distributed information divergence estimation over data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, 2014.

[2] S. Ganguly, M. Garafalakis, R. Rastogi, and K. Sabnani, "Streaming algorithms for robust, real-time detection of DDoS attacks," in *Proceedings of the 27th International Conference on Distributed Computing Systems*, ser. ICDCS, 2007.

[3] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, ser. RANDOM, 2002.

[4] D. M. Kane, J. Nelson, and D. P. Woodruff, "An optimal algorithm for the distinct elements problem," in *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS, 2010.

[5] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, ser. STOC, 1996.

[6] J. Misra and D. Gries, "Finding Repeated Elements," *Science of Computer Programming*, vol. 2, 1982.

[7] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ser. ICALP, 2002.

[8] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, 2005.

[9] G. Cormode, S. Muthukrishnan, and K. Yi, "Algorithms for distributed functional monitoring," *ACM Transactions on Algorithms*, vol. 7, 2011.

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM Journal on Computing*, vol. 31, 2002.

[11] P. B. Gibbons and S. Tirthapura, "Distributed streams algorithms for sliding windows," *Theory of Computing Systems*, vol. 37, 2004.

[12] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS, 2004.

[13] L. Zhang and Y. Guan, "Variance estimation over sliding windows," in *Proceedings of the 26h ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS, 2007.

[14] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro, "Identifying frequent items in sliding windows over on-line packet streams," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser. IMC, 2003.

[15] G. Cormode and K. Yi, "Tracking distributed aggregates over time-based sliding windows," in *Proceedings of the 24th International Conference on Scientific and Statistical Database Management*, ser. SSDBM, 2012.

[16] G. Cormode and S. Muthukrishnan, "What's hot and what's not: tracking most frequent items dynamically," *ACM Transactions Database Systems*, vol. 30, 2005.

[17] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis, "Sketch-based querying of distributed sliding-window data streams," *Proceedings of the VLDB Endowment*, vol. 5, 2012.

[18] E. Anceaume, Y. Busnel, N. Rivetti, and B. Sericola, "Identifying Global Icebergs in Distributed Streams," in *Proceedings of the 34th International Symposium on Reliable Distributed Systems*, ser. SRDS, 2015.

[19] S. Muthukrishnan, *Data streams: algorithms and applications*. Now Publishers Inc, 2005.

[20] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, 1979.

[21] N. Rivetti, Y. Busnel, and A. Mostefaoui, "Efficiently Summarizing Distributed Data Streams over Sliding Windows," Université de Nantes; Crest(Ensai); Inria Rennes, Tech. Rep. hal-01073877, Jun. 2015.

[22] CAIDA UCSD, "*DDoS Attack 2007* dataset," http://www.caida.org/data/passive/ddos-20070804_dataset.xml, Feb. 2010.

[23] CAIDA UCSD, "*Anonymized Internet Traces 2008* dataset," http://www.caida.org/data/passive/passive_2008_dataset.xml, Apr. 2008.