Efficiently Summarizing Distributed Data Streams over Sliding Windows

Nicolò Rivetti, Yann Busnel, Achour Mostefaoui

To cite this version:

Nicolò Rivetti, Yann Busnel, Achour Mostefaoui. Efficiently Summarizing Distributed Data Streams over Sliding Windows. 2014. hal-01073877v1

HAL Id: hal-01073877

https://hal.science/hal-01073877v1

Submitted on 10 Oct 2014 (v1), last revised 30 Jun 2015 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficiently Summarizing Distributed Data Streams over Sliding Windows

Nicolò Rivetti[2], Yann Busnel[1,2], and Achour Mostefaoui[2]

[1] Crest / Ensai, Rennes, France
[2] LINA / Université de Nantes, Nantes, France

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. In this paper, we consider the *sliding window functional monitoring* model and propose two different (on-line) algorithms that (ε, δ)-approximate the items frequency in the active window. They use a very small amount of memory with respect to the size of the window N and the number of distinct items n 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 with a communication cost of $O(\frac{k}{\varepsilon^2}\log\frac{1}{\delta}\log N)$ bits per window (where k is the number of nodes). Experiments on synthetic traces and real data sets validate the robustness and accuracy of our algorithms.

1 Introduction and Related Work

In large distributed systems it is most likely critical to gather various aggregates over data spread across multiple nodes. This can be modelled by a set of nodes, each observing a stream of items, collaborating to continuously track a function over the global distributed stream. For instance, current network management tools analyse the input streams of a set of routers to detect malicious sources or extract user behaviours [2,14,20]. The main goal is to evaluate such function minimizing the communication cost, the space used at each node, as well as the update and query time. Usually, 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 and number of distinct items n of the stream.

In the *data streaming* model, results have been shown for estimating the number of distinct data items in a stream [4,17], frequency moments [1], most frequent data items [18], frequency estimation [9,10] or information divergence over streams [2]. Cormode *et al.* [11] propose solutions for frequency moments estimation in the *functional monitoring* model. In most applications, computing such function from the inception of the distributed streams is useless. Only the most recent data may be relevant: the function has to be evaluated over a sliding

window of size N. Datar *et al.* [13] 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 [15] presented an algorithm matching the results of [13], based on the *waves* data structure. However they achieve constant processing time and provide extensions for distributed streams. Arasu and Manku [3] studied the problem of ε-approximating counts over sliding windows, presenting both a deterministic and randomized solutions achieving respectively $O(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon})$ and $O(\frac{1}{\varepsilon}\log\frac{\varepsilon}{\delta})$. In this model there are also works on variance [21], quantiles [3] and frequent items [16]. This model has also been applied to distributed streams [8,15]. Recently Cormode and Yi [12] presented an optimal solution based on the *backward/forward* paradigm for basic counting. Taking from [13], Gibbons and Tirthapura [15] propose also an optimal algorithm for counting distinct items. Both [8] and [12] provide an optimal solution for the heavy hitter problem with $O(\frac{k}{\varepsilon}\log_2(\varepsilon N)(\log_2 N+\log_2 n))$ bits of communication.

In this paper we extend a well-known algorithm for frequency estimation, namely the Count-Min sketch [10], in a windowed version. To our knowledge there is no prior work addressing this problem in the *sliding window functional monitoring* model, neither local nor distributed. Note that related problems are the identification or estimation of the most frequent items (usually called heavy hitters) and approximate counts. To be self-contained, we describe the computational model and some necessary background in Section 2. In Section 3 we propose two novel (ε,δ)-approximations, achieving respectively $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N+\log_2 n))$ and $O(\frac{1}{\tau\varepsilon}\log_2\frac{1}{\delta}(\log_2 N+\log_2 n))$ bits of space, where τ is an additional parameter limiting memory usage (see Section 3.4). Section 3.5 presents their application to distributed data streams with a communication cost of $O(\frac{k}{\varepsilon^2}\log_2\frac{1}{\delta}\log_2 N)$ bits per window. The efficiency of both algorithms is analysed and Section 4 presents an extended performance evaluation of the estimation accuracy of our algorithms, with both synthetic traces and real data sets.

2 Preliminaries and Background

2.1 Data Streaming Model

We present the computation model under which we analyse our algorithms and derive bounds: the functional monitoring model [11]. We consider a set of k nodes $u_1,\ldots,u_k$ where each node u_ℓ receives a massively long input stream σ_ℓ, that is, a sequence of elements $\langle a_1,a_2,\ldots,a_m\rangle$ called samples. Samples are drawn from a universe $[n]=\{1,2,\ldots,n\}$ of items. We have implicitly defined the size (or length) of the stream as m and the size of the universe (or number of distinct items) of the stream as n. Streams do not have to contain the same items and/or with the same number of occurrences. Each sequence can only be accessed in its given order (no random access). The targeted function must be computed in a single pass (on-line) and continuously. Furthermore, nodes are not aware of the length of the stream m. They may interact with a specific node called *coordinator*, but do not communicate with each other.

In order to reach these goals, we rely on randomized algorithms. Such an algorithm $\mathcal{A}$ is said to be an (ε, δ)-approximation of a function ϕ on σ if, for any sequence of items in the input stream σ, $\mathcal{A}$ outputs $\hat{\phi}$ such that $\mathbb{P}\{| \hat{\phi} - \phi | > \varepsilon\phi\} < \delta$, where ε, $\delta > 0$ are given as precision parameters. The interested reader is invited to read the extensive overview published by Muthukrishan in [19].

On the other hand, we are interested in the "recently observed" samples, that is, the *sliding window* model formalized by Datar *et al.* [13]. In this model samples arrive continuously and expire after exactly N steps. A step can be defined either as a time tick or a sample arrival, and we say that the window is time-based or count-based. In the following we deal with count-based sliding windows, where the window contains exactly N samples. With respect to sliding windows, *hopping* windows split the window into sub-windows; the window moves forward only when the most recent sub-window is completed. The challenge consists in achieving this computation in sub-linear space and communication in N and n.

2.2 Vanilla Count-Min Sketch

A relevant problem is the *frequency estimation* problem. A stream σ implicitly defines a frequency vector $\mathbf{f} = (f_1, \ldots, f_n)$, where f_j represents the number of occurrences of item j in σ. The goal is to provide an estimate $\hat{f}_j$ of f_j for each item $j \in [n]$ of the stream.

Cormode and Muthukrishnan have introduced in [10] the Count-Min sketch that provides, for each item j in a stream, an (ε, δ)-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 provide an efficient method to build large families of hash functions approximating the 2-universal property [7].

The Vanilla Count-Min sketch (Listing 2.1) consists of a two dimensional *count* matrix of size $c_1 \times c_2$, where $c_1 = \left\lceil \log_2 \frac{1}{\delta} \right\rceil$ and $c_2 = \left\lceil \frac{e}{\varepsilon} \right\rceil$. 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' minimizing the collisions impact. In other words, the algorithm returns, as $f_{j'}$ estimation, the cell associated with j' with the lowest value: $\hat{f}_{j'} = \min_{1\leq i\leq c_1}\{count[i][h_i(j')]\}$. The space complexity of this algorithm is $O(\frac{1}{\varepsilon} \log_2 \frac{1}{\delta}(\log_2 m + \log_2 n))$ bits, while update and query time complexities are $O(\log_2 1/\delta)$. Concerning its accuracy, the following quality bound holds: $\mathbb{P}\{| \hat{f}_j - f_j | \geq \varepsilon(m - f_j)\} \leq \delta$, while $f_j \leq \hat{f}_j$ is always true.

Listing 2.1: Count-Min Sketch

1: **init do**

```
 2:     count[1...c1][1...c2] ← 0⃗
 3:     Choose c1 independent hash functions h1...hc1 : [n] → [c2] from
 4: a 2-universal family.
 5: end init
 6: upon ⟨Sample | j⟩ do
 7:     for i = 1 to c1 do
 8:         count[i][hi(j)] ← count[i][hi(j)] + 1
 9:     end for
10: end upon
11: function getFreq(j)                                ▷ returns f̂j
12:     return min{count[i][hi(j)] | 1 ≤ i ≤ c1}
13: end function
```

3 Windowed Count-Min

The Count-Min algorithm solves brilliantly the frequency estimation problem in the data stream model. We propose two extensions to the sliding window model: Proportional and Splitter. Nevertheless, we first introduce two naive algorithms, which 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.

3.1 Perfect Windowed Count-Min

Perfect (Listing3.1) 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 2.2).

Listing 3.1: Perfect Windowed Count-Min

```
 1: init do
 2:     count[1...c1][1...c2] ← 0⃗
 3:     Choose c1 independent hash functions h1...hc1 : [n] → [c2] from
 4: a 2-universal family.
 5:     samples ← ∅ queue of samples
 6: end init
 7: upon ⟨Sample | j⟩ do
 8:     for i = 1 to c1 do
 9:         count[i][hi(j)] ← count[i][hi(j)] + 1
10:     end for
11:     enqueue j in samples
12:     if | samples |> N then
```

```
13:         j' ← dequeue from samples
14:         for i = 1 to c_1 do
15:             count[i][h_i(j')] ← count[i][h_i(j')] − 1
16:         end for
17:     end if
18: end upon
19: function GETFREQ(j)                                  ▷ returns f̂_j
20:     return min{count[i][h_i(j)] | 1 ≤ i ≤ c_1}
21: end function
```

Theorem 1. PERFECT *is an* (ε, δ)*-approximation of the frequency estimation problem in the sliding windowed functional monitoring model where* $\mathbb{P}\{| \hat{f}_j - f_j | \geq \varepsilon(N - f_j)\} \leq \delta$*, while* $f_j \leq \hat{f}_j$ *is always true.*

Proof. Since the algorithm stores the whole previous window, it knows exactly which sample expires in the current step and can decrease the associated counters in the *count* matrix. Then PERFECT provides an estimation with the same error bounds of a VANILLA COUNT-MIN executed on the last N samples of the stream. □

Theorem 2. PERFECT *space complexity is* $O(N)$ *bits, while update and query time complexities are* $O(\log_2 \frac{1}{\delta})$*.*

Proof. The algorithm stores N samples, which leads to a space complexity of $O(N)$ bits, assuming that $N = \omega(\frac{1}{\varepsilon} \log_2 \frac{1}{\delta}(\log_2 N + \log_2 n))$. An update requires to enqueue and dequeue two samples ($O(1)$), and to manipulate a cell on each row. Thus the update time complexity is $O(\log_2 \frac{1}{\delta})$. A query requires to look up a cell for each row of the *count* matrix: the query time complexity is $O(\log_2 \frac{1}{\delta})$. □

3.2 Simple Windowed Count-Min

SIMPLE (Listing3.2) 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.

Listing 3.2: SIMPLE WINDOWED COUNT-MIN

```
1: init do
2:     count[1...c_1][1...c_2] ← 0⃗
3:     Choose c_1 independent hash functions h_1...h_{c_1} : [n] → [c_2] from
4: a 2-universal family.
5:     m' ← 0
6: end init
7: upon ⟨Sample | j⟩ do
```

```
 8:     if m' = 0 then
 9:         count[1...c1][1...c2] ← 0⃗
10:     end if
11:     for i = 1 to c1 do
12:         count[i][h_i(j)] ← count[i][h_i(j)] + 1
13:     end for
14:     m' ← m' + 1  mod N
15: end upon
16: function getFreq(j)                                  ▷ returns f̂_j
17:     return min{count[i][h_i(j)] | 1 ≤ i ≤ c1}
18: end function
```

Theorem 3. Simple *space complexity is* $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n))$ *bits, while update and query time complexities are* $O(\log_2\frac{1}{\delta})$.

Proof. The algorithm uses a counter of size $O(\log_2 N)$ and a matrix of size $c_1 \times c_2$ ($c_1 = \lceil\log_2\frac{1}{\delta}\rceil$ and $c_2 = \lceil\frac{e}{\varepsilon}\rceil$) of counters of size $O(\log_2 N)$. In addition, for each row it stores a hash-function. Then the space complexity is $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n))$ bits. An update requires to hash a sample, then retrieve and increase a cell for each row, thus the update time complexity is $O(\log_2\frac{1}{\delta})$. We consider the cost of resetting the matrix ($O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta})$) negligible since it is done only once per window. A query requires to hash a sample and retrieve a cell for each row: the query time complexity is $O(\log_2\frac{1}{\delta})$ □

3.3 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 increases 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 (Listing3.3), 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 ($\forall i \in [c_1], count[i, h_i(j)] \leftarrow count[i, h_i(j)] + 1$) 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.

Listing 3.3: Proportional Windowed Count-Min

```
 1: init do
 2:     count[1...c1][1...c2] ← 0⃗
```

3: Choose c_1 independent hash functions $h_1 \dots h_{c_1} : [n] \to [c_2]$ from
4: a 2-universal family.
5: $snapshot[1 \dots c_1][1 \dots c_2] \leftarrow \vec{0}$
6: $m' \leftarrow 0$
7: **end init**
8: **upon** $\langle Sample \mid j \rangle$ **do**
9: **if** $m' = 0$ **then**
10: **for** $i_1 = 1$ to c_1 and $i_2 = 1$ to c_2 **do**
11: $snapshot[i_1][i_2] \leftarrow \frac{count[i_1][i_2]}{N}$
12: **end for**
13: **end if**
14: **for** $i_1 = 1$ to c_1 and $i_2 = 1$ to c_2 **do**
15: **if** $h_{i_1}(j) = i_2$ **then**
16: $count[i_1][i_2] \leftarrow count[i_1][i_2] + 1$
17: **end if**
18: $count[i_1][i_2] \leftarrow count[i_1][i_2] - snapshot[i_1][i_2]$
19: **end for**
20: $m' \leftarrow m' + 1 \mod N$
21: **end upon**
22: **function** GETFREQ(j) ▷ returns $\hat{f}_j$
23: **return** round$\{\min\{count[i][h_i(j)] \mid 1 \leq i \leq c_1\}\}$
24: **end function**

Theorem 4. PROPORTIONAL *space complexity is* $O(\frac{1}{\varepsilon} \log_2 \frac{1}{\delta}(\log_2 N + \log_2 n))$ *bits. Update and query time complexities are* $O(\frac{1}{\varepsilon} \log_2(1/\delta))$ *and* $O(\log_2 \frac{1}{\delta})$.

Proof. The algorithm stores a *count* and a snapshot matrix, as well as a counter of size $O(\log_2 N)$. Then the space complexity is $O(\frac{1}{\varepsilon} \log_2 \frac{1}{\delta}(\log_2 N + \log_2 n))$ bits. An update require to look up all the cells of both the *count* and *snapshot*, thus the update time complexity is $O(\frac{1}{\varepsilon} \log_2 \frac{1}{\delta})$. A query requires to hash a sample and retrieve a cell for each row: the query time complexity is $O(\log_2 \frac{1}{\delta})$ □

3.4 Splitter Windowed Count-Min

As one could observe in Section 4, PROPORTIONAL provides quite good performances. However the frequency distribution of the previous window is *averagely* removed from the current window. Thus 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 analysing the update rate of each cell in the *count* matrix. To 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. **??** presents the global behaviour 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 (Listing3.5) 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 behaviours. To avoid this, we introduced τ that sets the minimal length of a sub-cell before taking the sub-cell into account in the decision process.

In more details, when SPLITTER reads sample j, for each cell of *count* it retrieves the oldest sub-cell in the queue, denoted $first$ (Line 10). If $first$was active precisely N steps ago (Line 11), SPLITTER computes the rate at which $first$ has been incremented while it was active (Line 12). This value is then subtracted from the cell counter v (Line 13) and from $first$ counter (Line 14). Having retracted what happened N steps ago, $first$ moves forward increasing its *init* timestamp (Line 15). Finally, $first$ is removed if it has expired (Lines 16 and 17).

The next part handles the update of the cells associated with item j. For each of them (Line 20), SPLITTER increases the cell counter v (Line 21) and retrieves the current sub-cell, denoted *last* (Line 22). **(a)** If *last* does not exist, it creates and enqueues a new sub-cell (Lines 24 to 28). **(b)** If *last* has not reached the minimal size to be evaluated (Line 29), *last* is updated (Lines 30 and 31). **(c)** If not, it retrieves the predecessor of *last*: *pred* (Line 33). **(c.i)** If *pred* exists and the amount of information lost by merging is lower than the threshold μ (Line 34), SPLITTER merges *last* into *pred* and renews *last* (Lines 35 to 39). **(c.ii)** Otherwise it creates and enqueues a new sub-cell (Lines 41 to 45), *i.e.*, it *splits* the cell.

Listing 3.4: SPLITTER WINDOWED COUNT-MIN

```
1:  init do
2:      count[1...c1][1...c2] ← ⟨∅,0⟩→                  ▷ the set is a queue
3:      Choose c1 independent hash functions h1...hc1 : [n] → [c2] from
4:  a 2-universal family.
5:      m' ← 0
6:  end init
7:  upon ⟨Sample | j⟩ do
8:      for i1 = 1 to c1 and i2 = 1 to c2 do
9:          ⟨queue, v⟩ ← count[i1][i2]
10:         first ← head of queue               ▷ retrieves the oldest sub-cell
```

11: **if** $\exists first \wedge first_{init} = m' - N$ **then** ▷ if the sub-cell was active
12: $v' \leftarrow \frac{first_{counter}}{first_{last} - first_{init} + 1}$
13: $v \leftarrow v - v'$
14: $first_{counter} \leftarrow first_{counter} - v'$
15: $first_{init} \leftarrow first_{init} + 1$
16: **if** $first_{init} > first_{last}$ **then**
17: remove $first$ from queue
18: **end if**
19: **end if**
20: **if** $h_{i_1}(j) = i_2$ **then** ▷ handle update for sample j
21: $v \leftarrow v + 1$
22: $last \leftarrow$ bottom of $queue$ ▷ retrieves the newest sub-cell
23: **if** $\nexists last$ **then**
24: $last \leftarrow$ new sub-cell
25: $last_{init} \leftarrow m'$
26: $last_{last} \leftarrow m'$
27: $last_{counter} \leftarrow 1$
28: enqueues new in $queue$
29: **else if** $last_{counter} < \frac{\tau N}{c_2}$ **then**
30: $last_{last} \leftarrow m'$
31: $last_{counter} \leftarrow last_{counter} + 1$
32: **else**
33: $pred \leftarrow$ predecessor of the last in $queue$
34: **if** $\exists pred \wedge \text{ERROR}(pred, last) \leq \mu$ **then** ▷ merge check
35: $pred_{last} \leftarrow last_{last}$
36: $pred_{counter} \leftarrow pred_{count} + last_{count}$
37: $last_{init} \leftarrow m'$
38: $last_{last} \leftarrow m'$
39: $last_{counter} \leftarrow 1$
40: **else**
41: $new \leftarrow$ new sub-cell
42: $new_{init} \leftarrow m'$
43: $new_{last} \leftarrow m'$
44: $new_{counter} \leftarrow 1$
45: enqueues new in $queue$ ▷ *splits* the cell
46: **end if**
47: **end if**
48: **end if**
49: $count[i_1][i_2] \leftarrow \langle queue, v \rangle$
50: **end for**
51: $m' \leftarrow m' + 1$
52: **end upon**
53: **function** GETFREQ(j) ▷ returns $\hat{f}_j$
54: **return** round$\{\min\{count[i][h_i(j)].v \mid 1 \leq i \leq c_1\}\}$
55: **end function**

Listing 3.5: SPLITTER WINDOWED COUNT-MIN error function

```
1: function ERROR(pred, last)
2:     freq_pred ← pred_count / (last_init − pred_init)
3:     freq_last ← last_count / (last_init − last_init + 1)
4:     if freq_last > freq_pred then
5:         return freq_last / freq_pred
6:     else
7:         return freq_pred / freq_last
8:     end if
9: end function
```

Lemma 1. *The total number s of splits (number of sub-cell spawned to track distribution changes) is* $O(\frac{1}{\varepsilon\tau}\log_2\frac{1}{\delta})$.

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 $s = O(\frac{1}{\varepsilon\tau}\log_2\frac{1}{\delta})$. □

Theorem 5. SPLITTER *space complexity is* $O(\frac{1}{\tau\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n))$ *bits, while update and query time complexities are* $O(1/\varepsilon\log_2 1/\delta)$ *and* $O(\log_2\frac{1}{\delta})$.

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[3] of size $O(\log_2 N)$ bits. Without any split and considering that all cells have bootstrapped, the initial space complexity is $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n))$ bits. Each split costs two additional timestamps and a counter (size of a sub-cell). Let s be the number of splits, then $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n) + s\log_2 N)$ bits. Lemma 1 establishes the following space complexity bound: $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n) + \frac{1}{\varepsilon\tau}\log_2\frac{1}{\delta}\log_2 N)$ bits.

Each update requires to retrieve and manipulate the $first$ sub-cell of all the *count* matrix cells. Then, for each row it has to retrieve a cell and manipulate its *last* and *pred* sub-cells. Thus, the update time complexity is $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta})$. Each query requires to lookup one cell by row of the *count* matrix. Then, the query time complexity is again $O(\log_2\frac{1}{\delta})$. □

Note that the space complexity can be reduced by removing the cell counter v. However, the query time would increase since this counter must be reconstructed summing all the sub-cell counters. One can argue that sub-cell creations and destructions cause memory allocations and disposals. However, we believe that one can avoid wild memory usage leveraging the sub-cell creation patterns, either through a smart memory allocator or a memory aware data structure.

Finally, Table 1 summarizes the space, update and query complexities of the presented algorithms.

[3] Note that, for sake of clarity, timestamps are of size $O(\log_2 m)$ bits in the pseudo-code. However, counters of size $O(\log_2 N)$ bits are sufficient.

Table 1: Complexities comparison

Algorithm	Space (bits)	Update time	Query time
Vanilla Count-Min	$O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 m + \log_2 n))$	$O(\log_2\frac{1}{\delta})$	$O(\log_2\frac{1}{\delta})$
Perfect	$O(N)$	$O(\log_2\frac{1}{\delta})$	$O(\log_2\frac{1}{\delta})$
Simple	$O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n))$	$O(\log_2\frac{1}{\delta})$	$O(\log_2\frac{1}{\delta})$
Proportional	$O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n))$	$O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta})$	$O(\log_2\frac{1}{\delta})$
Splitter	$O(\frac{1}{\tau\varepsilon}\log_2\frac{1}{\delta}(\log_2 N + \log_2 n))$	$O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta})$	$O(\log_2\frac{1}{\delta})$

3.5 Distributed Count-Min

Note 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 [12], we can define the DistCM algorithm, which sends the *count* matrix to the coordinator each εN samples. DistCM can be applied to the four aforementioned windowed extensions of Vanilla Count-Min, resulting in a distributed frequency (ε, δ)-approximation in the *sliding windowed distributed functional monitoring* model. Due to space constraints, proofs of the following theorems are available in Appendix **??**.

Theorem 6. DistCM *communication complexity is* $O(\frac{k}{\varepsilon^2}\log_2\frac{1}{\delta}\log_2 N)$ *bits per window.*

Proof. In each window and for each node u_ℓ ($\ell \in [k]$), DistCM sends the *count* matrix at most $\frac{N}{\varepsilon N} = \frac{1}{\varepsilon}$ times. Thus the communication complexity is $O(\frac{k}{\varepsilon^2}\log_2\frac{1}{\delta}\log_2 N)$ bits per window. □

Theorem 7. DistCM *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* $k\varepsilon N$*.*

Proof. Similarly to [12], the coordinator misses for each node u_ℓ ($\ell \in [k]$) at most the last εN increments. Then, the global $\overline{count}$ cells cannot fall behind by more than $k\varepsilon N$ increments. Thus DistCM introduces at most an additive error of $k\varepsilon N$. □

3.6 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 arrival.

In each algorithm it is possible to split the update code into a subroutine increasing the *count* matrix and a subroutine 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 (*i.e.* step). Note that time-stamps have to be updated using the current 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.

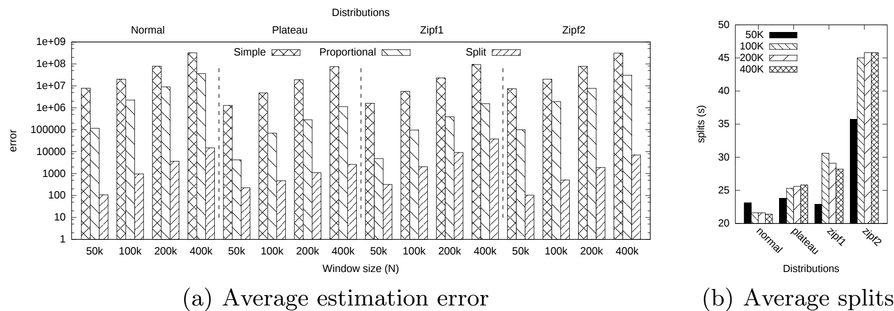
4 Performance Evaluation

This section provides the performance evaluation of our algorithms. We have conducted a series of experiments on different types of stream and parameter settings. To check 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 1-st and 10-th deciles to avoid outliers.

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 behaviour of the algorithms in the following windows does not change. 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 $\varepsilon = 0.01$, meaning $c_2 = \lceil \frac{e}{0.01} \rceil = 28$ columns. Most of the time, the *count* matrix has several rows. However, analysing results using multiple rows requires taking into account the interaction between the hash functions. If not specified, for 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 width w, a period p and a number of repetitions 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 $r = 4$.



(a) Average estimation error (b) Average splits

Fig. 1: Results for different window sizes: $N \in \{50k, 100k, 200k, 400k\}$

We evaluate the performance by generating families of synthetic streams, following five distributions: **(i) Uniform**: uniform distribution; **(ii) Normal**: truncated standard normal distribution; **(iii) Zipf1**: Zipfian distribution with $\alpha = 1.0$; **(iv) Zipf2**: Zipfian distribution with $\alpha = 2.0$; and **(v) Plateau**: a distribution where the probabilities are divided in two levels: a set of w items is 100 times more probable than the other $n - w$ samples.

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 analysed in Section 4.2. Given these parameters, we have an upper bound of $s = 560$ spawned sub-cells.

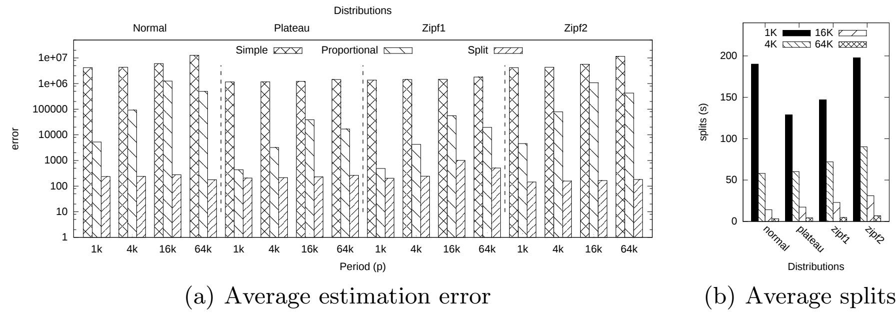
Finally, the accuracy metric used in our evaluation is the mean squared error of the frequency estimation of all n items returned by the algorithms with respect to PERFECT, that is $\sum_{j \in [n]} (\hat{f}_j^{\text{PERFECT}} - \hat{f}_j^{\text{ALGO}})^2 / n$. We refer to this metric as *estimation error*.

We also evaluate the additional space used by SPLITTER, due to merge and split mechanism, through the number of splits s.

4.1 Comparing the performance of all algorithms

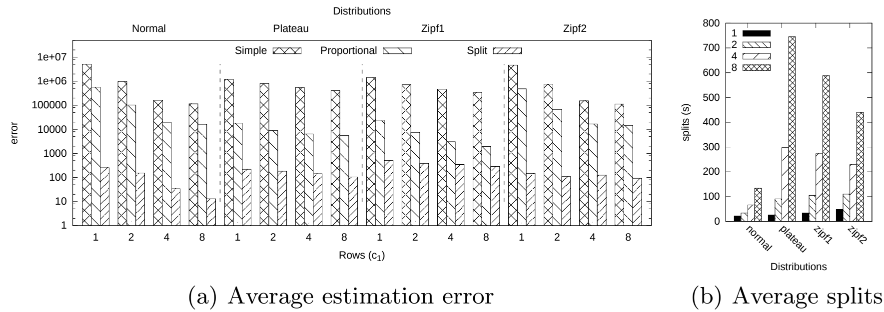
Window sizes Figure 1(a) presents the estimation error of the SIMPLE, PROPORTIONAL and SPLITTER algorithms considering the Normal, Plateau, Zipf1 and Zipf2 distributions, with $N = 50,000$ (so $m = 150,000$), $N = 100,000$ (so $m = 300,000$), $N = 200,000$ (so $m = 600,000$) and $N = 400,000$ (so $m = 1,200,000$). Note that the y-axis (*error*) is in logarithmic scale and error values are averaged over the whole stream. SIMPLE is always the worst (more than 10^7 in average), followed by PROPORTIONAL (roughly 2.4×10^6 in average), while SPLITTER is always the best (less than 3.7×10^4). The error estimation of SIMPLE, PROPORTIONAL and SPLITTER increases in average $3.7\times$, $8.9\times$ and $4.2\times$ respectively for each 2-fold increase of N.

Figure 1(b)give the number of splits SPLITTER spawned in average to keep up with the distribution changes. The number of splits is globally unaffected by N since the ratio N/p remain constant: in average, there are 30 splits, while the standard deviation over all experiments is 2.2.



(a) Average estimation error (b) Average splits

Fig. 2: Results for different periods: $p \in \{1k, 4k, 16k, 64k\}$



(a) Average estimation error (b) Average splits

Fig. 3: Results for different number of rows: $c_1 \in \{1, 2, 4, 8\}$

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 2. Again, Splitter is always the best (at most 10^3), followed by Proportional (roughly 2×10^5 in average), while Simple is always the worst (more than 10^6). Simple grows slowly but continuously (in average 2 times from $1,000$ to $64,000$) because slower shifts cast all the error on less items, resulting in a larger mean squared error. The same phenomenon causes also the Proportional trend from $1,000$ to $16,000$. 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 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.

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 3 presents the estimation error and splits for $c_1 = 1$ ($\delta = 0.5$),

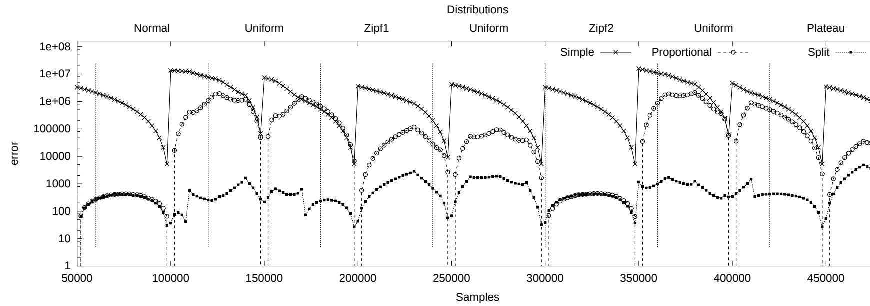


Fig. 4: Estimation error with multiple distributions

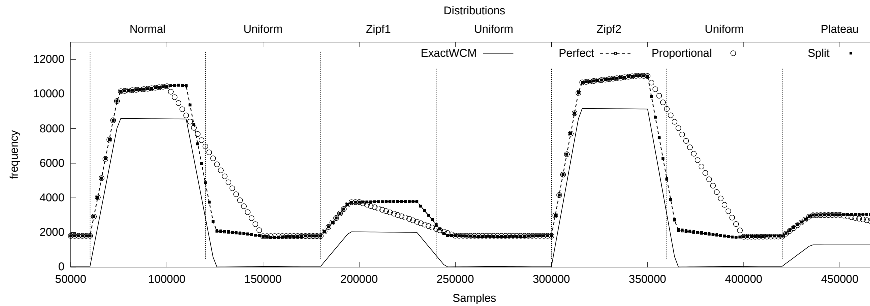


Fig. 5: Frequency estimation of item 0 with multiple distributions

$c_1 = 2$ ($\delta = 0.25$), $c_1 = 4$ ($\delta = 0.0625$) and $c_1 = 8$ rows ($\delta = 0.004$). Increasing the number of rows do enhance the accuracy of the algorithms. However, Simple does not better than 10^5, followed by Proportional, roughly 8.5×10^4 in average, and Splitter has the lowest error, at most 513. For each distribution shift, $2w$ item 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, the number of splits growth is not quadratic: in average it increases by $2.4\times$ for each 4-fold increase of c_1.

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, Zipf1, Uniform, Zipf2, Uniform, Plateau. The streams is of length $m = 480,000$. Note that the distribution shifts and swaps are not synchronized with the window swaps ($N = 50,000$ samples).

Figure 4 presents the estimation error evolution as the stream unfolds. Splitter error does not exceed 5×10^3 (in average 718), while Proportional goes up to 2×10^6 (in average 3×10^5) and Simple does not better than 5×10^3 (in average 2.6×10^6). Since at the beginning of each window Simple resets its *count* matrix, there is a periodic behaviour: the error burst when a window starts and shrinks towards the end. In the 1-st (0 to $50,000$) and in the 6-th windows

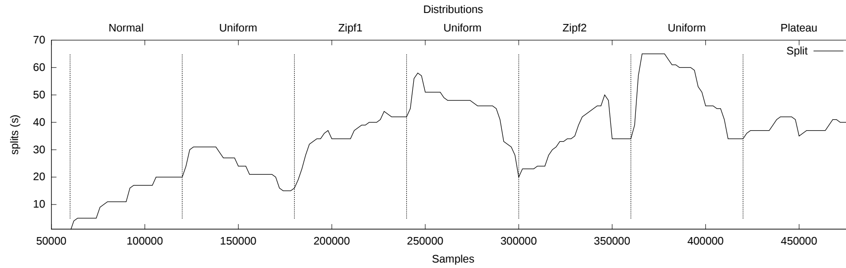


Fig. 6: Number splits s with multiple distributions

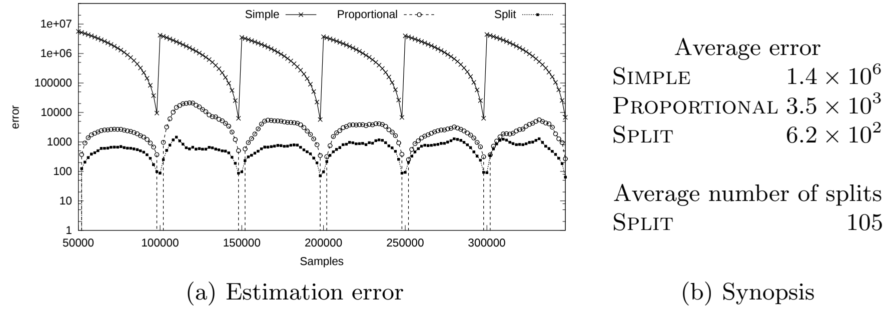
(250, 000 to 300, 000) the distribution does not change over time (shifting Uniform has no effect). This means that SPLITTER does not capture more information than PROPORTIONAL, thus they provide the same estimations in the 2-nd (50, 000 to 100, 000) and the 7-th windows (300, 000 to 350, 000).

Figure 5 presents the value of f_0 and its estimations over time (for clarity SIMPLE is omitted). The plain line is f_0, which also reflects the distribution changes. The plots for PERFECT and SPLITTER are overlapping, this is why only blackened squares are visible. Except for the error introduced by the COUNT-MIN approximation, they both follow the f_0 shape precisely. Item 0 occurrence probability changes significantly in the following intervals: $[60k, 75k]$, $[180k, 195k]$, $[300k, 315k]$ and $[420k, 435k]$. PROPORTIONAL fails to follow the f_0 trend in the windows following those intervals, namely the 3-rd, 5-th, 8-th and 10-th, since it is unable to correctly asses the previous window distribution.

Finally, Figure 6 presents the number of splits s. There are in average 31 and at most 66 splits (while s upper bound is 560). Splits decrease when the distribution does not change (in the Uniform intervals): 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, meaning that the change is detected. This pattern is clearly visible in the 2-nd window.

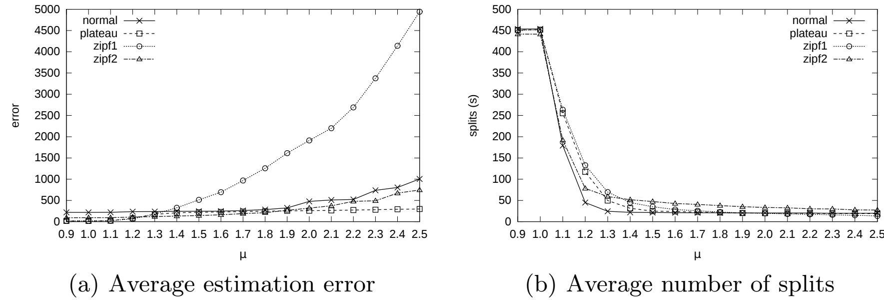
Real DDoS trace We have retrieved the CAIDA "DDoS Attack 2007" [6] and "CAIDA Anonymized Internet Traces 2008" [5] datasets, interleaved them and retained the first 350, 000 samples (*i.e.*, the DDoS attack beginning). The stream has $n = 4.9 \times 10^4$ distinct items. The item representing the DDoS target has a frequency proportion equal to 0.09, while the following most frequent item frequency proportion is 0.004. Figure 7(a) presents the estimation error evolution. 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[4]. We can see trends similar to the previous test, however the estimation provided by PROPORTIONAL is closer to SPLITTER since the stream changes much less over time. SIMPLE does not better than 5.7×10^3,

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



(a) Estimation error (b) Synopsis

Fig. 7: Results for the DDOS trace



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

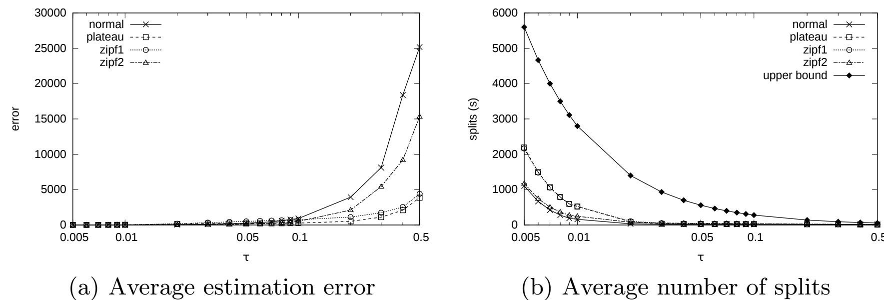
Fig. 8: Performance comparison with $\tau = 0.05$ and $\mu \in \{0.9, 2.5\}$

while PROPORTIONAL and SPLITTER do not exceed 2.1×10^4 and 1.4×10^3 respectively. As for the splits, there are at most 154 splits with an average of 105 splits. Figure 7(b) resumes the average values over the whole stream.

4.2 Impact of the Splitter parameters

Figure 8 presents the estimation error and the number of splits with $\mu \in \{0.9, 2.5\}$ and $\tau = 0.05$. As expected, the estimation error grows with μ. Zipf1 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, roughly 19% less than the upper bound. From $\mu = 1.0$ to 1.3, the average falls down to 51, reaching 20 at $\mu = 2.5$.

Figure 9 presents the estimation error and splits with $\tau \in \{0.005, 0.5\}$ and $\mu = 1.5$. Note that the x-axis (τ) is logarithmic. As for μ, increasing τ increases the estimation error: the average starts at 4 ($\tau = 0.005$), reaches 610 at $\tau = 0.1$ and grows up at $12,198$ ($\tau = 0.5$). 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 ($\tau = 0.5$). Figure 9(b) presents also the theoretical upper bound.



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

Fig. 9: Performance comparison with $\mu = 1.5$ and $\tau \in \{0.005, 0.5\}$

The trend in all 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 to minimize the error and the splits. This optimal value could either be independent from the stream distribution or computed based on the recent behaviour of the algorithm and some constraints provided by the user.

5 Conclusion and Future Work

We have presented two (ε, δ)-approximations for the frequency estimation problem in the sliding window functional monitoring model: Proportional and Splitter. We have proven that their space complexities are $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta}(\log_2 N+\log_2 n))$ and $O(\frac{1}{\tau\varepsilon}\log_2\frac{1}{\delta}(\log_2 N+\log_2 n))$ bits respectively, while their update and query time complexities are $O(\frac{1}{\varepsilon}\log_2\frac{1}{\delta})$ and $O(\log_2\frac{1}{\delta})$. Leveraging the *sketch* property, we have shown how they can be applied to distributed data streams with a communication cost of $O(\frac{k}{\varepsilon^2}\log_2\frac{1}{\delta}\log_2 N)$ bits per window. However, we believe there is still room for improvement.

We have performed an extensive performance evaluation showing the accuracy of both algorithms in the face of real world traces and of specifically tailored adversarial synthetic traces. We have also studied the impact of the two additional parameters of Splitter: τ and μ.

From these results, we are looking forward to 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. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of the 28th Annual ACM Symposium on Theory of Computing*, STOC '96, 1996.

2. E. Anceaume and Y. Busnel. A distributed information divergence estimation over data streams. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):478–487, 2014.
3. A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proc. of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, 2004.
4. Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. of the 6th International Workshop on Randomization and Approximation Techniques*, RANDOM '02, 2002.
5. CAIDA UCSD. "Anonymized Internet Traces 2008" Dataset. `http://www.caida.org/data/passive/passive_2008_dataset.xml`, Apr. 2008.
6. CAIDA UCSD. "DDoS Attack 2007" Dataset. `http://www.caida.org/data/passive/ddos-20070804_dataset.xml`, Feb. 2010.
7. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.
8. H.-L. Chan, T.-W. Lam, L.-K. Lee, and H.-F. Ting. Continuous monitoring of distributed data streams over a time-based sliding window. *Algorithmica*, 62(3-4):1088–1111, 2012.
9. M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, 2002.
10. G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
11. G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. *ACM Trans. on Algorithms*, 7(2):21:1–21:20, 2011.
12. G. Cormode and K. Yi. Tracking distributed aggregates over time-based sliding windows. In *Proc. of the 24th International Conference on Scientific and Statistical Database Management*, SSDBM'12, 2012.
13. M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.
14. S. Ganguly, M. Garafalakis, R. Rastogi, and K. Sabnani. Streaming algorithms for robust, real-time detection of ddos attacks. In *Proc. of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, 2007.
15. P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. *Theory of Computing Systems*, 37(3):457–478, 2004.
16. 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 *Proc. of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, 2003.
17. D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proc. of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, 2010.
18. J. Misra and D. Gries. Finding Repeated Elements. *Science of Computer Programming*, 2:143–152, 1982.
19. S. Muthukrishnan. *Data streams: algorithms and applications*. Now Pub. Inc, 2005.
20. T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu. What happened in my network: mining network events from router syslogs. In *Proc. of the 10th ACM Conference on Internet measurement*, IMC '10, 2010.
21. L. Zhang and Y. Guan. Variance estimation over sliding windows. In *Proc. of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, 2007.