Light Agents Searching for Hot Information

Dariusz R. Kowalski[1] and **Dominik Pajak**[2]
[1]School of Computer and Cyber Sciences, Augusta University
[2]Wrocław University of Science and Technology
dkowalski@augusta.edu, dominik.pajak@pwr.edu.pl

Abstract

Agent-based crawlers are commonly used in network maintenance and information gathering. In order not to disturb the main functionality of the system, whether acting at nodes or being in transit, they need to operate online, perform a single operation fast and use small memory. They should also be preferably deterministic, as crawling agents have limited capabilities of generating a large number of truly random bits. We consider a system in which an agent receives an update, typically an insertion or deletion, of some information upon visiting a node. On request, the agent needs to output hot information, i.e., with the net occurrence above certain frequency threshold. A desired time and memory complexity of such agent should be poly-logarithmic in the number of visited nodes and inversely proportional to the frequency threshold. Ours is the first such agent with rigorous analysis and a complementary almost-matching lower bound.

1 The Model and the Problem

Finding elements occurring above certain frequency φ, so called hot items, is one of the fundamental tools in mining online streams and histogram study, c.f., [Ioannidis and Christodoulakis, 1993; Ioannidis and Poosala, 1995]. It can also be applied in data warehousing, data mining and information retrieval, decision support systems, databases, caching, load balancing, network management, anomaly detection, c.f., [Demaine *et al.*, 2002; Fang *et al.*, 1998; Gibbons and Matias, 1999; Karp *et al.*, 2003].

We consider a stream of operations, also called transactions, involving elements in the universe N of size n. We do not limit the distribution of elements in the stream – they could be created arbitrarily, even by an online adversary, which aims at "fooling" the agent processing the stream aiming to find hot elements. Each operation involves a single element $x \in N$ and could be either an insertion or deletion of this element (to/from some large data repository). Observe that if a large $\Theta(n)$ space is allowed at an agent, then a simple heap data structure could process each insertion or deletion operation in $O(\log\ n)$ time, and find the hot items in $O(\frac{1}{\epsilon}\log\ n)$ time, for any stream, c.f.,[Aho *et al.*, 1983]. However, in case of large universe N (Big Data), such a solution is not practical. Therefore, for more than 20 years the research in this area focused on finding a summary data structure, of sublinear (in n) size and processing/enlisting time.

There is, however, a subtle twist – could sublinear algorithms return all and only hot elements? [Cormode and Muthukrishnan, 2005b] showed that enlisting *all and only* hot elements is impossible with sublinear memory $o(n)$. (They were inspired by a seminal paper [Alon *et al.*, 1996] proving that estimating highest frequency is impossible in sublinear memory $o(n)$.) This also applies to randomized algorithms: any algorithm which guarantees outputting all and only hot elements with probability at least $1-\sigma$, for some constant σ, must also use $\Omega(n)$ memory. This generalization uses a related result on the Index problem in communication complexity, c.f., [Kushilevitz and Nisan, 1997]. This argument implies that, if we are to use less than $\Theta(n)$ memory, then we must sometimes output items with frequency smaller than φ. Therefore, the main challenge is:

To design an efficient (light) deterministic online algorithm (agent) processing any stream of operations and, upon request, listing all elements of frequency at least φ and no element of frequency at most $\varphi-\epsilon$. Light, or well-scalable, agent means that it should process an operation or output an element in time at most polylogarithmic in n (i.e., $\log^c n$ for some constant $c>0$), as $n=|N|$ could be very large, while using memory linear in $1/\epsilon$ and polylogarithmic in n. Light algorithms have already been designed in some cases:

- When randomness is allowed, c.f., [Cormode and Muthukrishnan, 2005b]; however, false positives and false negatives are possible; also, it is not known if the result holds against adversarial streaming, i.e., if the adversary decides on consecutive elements in the stream online, seeing the past choices of the algorithm,
- When second pre-processing or processing in larger batches (so called window-based) is allowed, c.f., [Misra and Gries, 1982; Lin *et al.*, 2005] however, such agents are not pure online.

Our results. In this work (Section 2) we design a deterministic algorithm that overcomes all of the abovementioned obstacles: it is fully online (it does not go backwards or look ahead when processing current stream location), it does not use any random bits, and it works for arbitrary streams even created by an online adaptive adversary. It handles both insertions and deletions. Finally, it is also light and well-scalable (as we analyze formally in Section 3), in the sense that it uses only polylogarithmic time per operation and returning a hot element while using only $O(\frac{\log^3 n}{\epsilon})$ local memory. This memory space is close to

Type of algorithm	Time per item	Memory	Operations	Reference
Deterministic – two passes	$O(\frac{1}{\varphi}\log(\varphi n))$ *	$O(\frac{1}{\epsilon})$	insert only	[Misra and Gries, 1982]
Deterministic	$O(\frac{1}{\varphi}\log(\varphi n))$ †	$O(\frac{\log(n/\epsilon)}{\epsilon})$	insert only	[Manku and Motwani, 2002]
Randomized LV	$O(1)$ expected	$O(\frac{1}{\epsilon})$	insert only	[Demaine *et al.*, 2002]
Randomized MC – approx	$O(\log\frac{1}{\sigma})$	$O(\frac{\log n}{\varphi\varepsilon^2})$	insert only	[Charikar *et al.*, 2002]
Randomized MC – approx	$O(\log n\cdot\log\frac{1}{\varphi\sigma})$	$O\left(\frac{\log n\cdot\log\frac{1}{\varphi\sigma}}{\varepsilon}\right)$	insert & delete	[Cormode and Muthukrishnan, 2005b]
Deterministic	$O\left(\frac{\log_{1/(\varphi\epsilon)} n\cdot\log n}{\varphi\epsilon}\right)$	$O\left(\frac{\log^3 n}{\varphi^2\epsilon^2}\right)$	insert & delete	[Ganguly and Majumder, 2007]
Deterministic	$O(\text{ polylog } n)$	$O(\frac{\log^3 n}{\varepsilon})$	insert & delete	this paper

Table 1: Performance of best algorithms finding hot elements. LV and MC denote Las Vegas and Monte Carlo solutions, resp., while approx mean only approximated solutions. * shows additionally that an *amortized* processing time is $O(\log\frac{1}{\varphi})$, while in † the amortized time is $O(\log(n\varphi))$, as discussed in [Cormode and Muthukrishnan, 2005b]. Amortized time denotes the total time for the whole stream processing divided by the number of processed operations, and is a weaker measure that worst-case complexity considered in this work. In the results of the existing papers a notation k was sometimes used that corresponds to $\varphi=1/(k+1)$. Notation σ in the existing papers denotes the probability of failure. The result of [Misra and Gries, 1982] and [Demaine *et al.*, 2002] consider the special case, where $\epsilon=\varphi$.

optimal, as we show in the proof of a lower bound in Section 4. Table 1 compares performance of our agent with most relevant previous work. Finally, we discuss possible extensions (including multi-agent parallelization) and open problems in Section 5.

Model and problem. Consider an incoming very long stream of operations on elements of a very large universe N of size $n=|N|$, where each operation is of a form $\mathsf{op}(x)$, for $\mathsf{op}\in\{\mathsf{insert},\mathsf{delete}\}$ and $x\in N$. We assume that each operation contains a $O(\log n)$-bit identifier of an element x involved; we will be using "element" and "item" interchangeably throughout this paper. The *net occurrence* of an item $x\in N$ in step t, denoted by $n_t(x)$, is the number of insertions of x minus the number of deletions of x in the first t operations of the stream. The *frequency* of element x in step t is denoted by $f_t(x)=n_t(x)/t$. The stream satisfies Basic Integrity Constraint, as defined in [Cormode and Muthukrishnan, 2005b]: in each round a frequency of any element is non-negative, in particular, the number of deletions never exceeds the number of insertions of an element.

The problem of finding hot elements (also called frequent elements) is parametrized by $1>\varphi>\epsilon>0$; we denote it by finding (φ,ϵ)-hot elements. The objective is to design an agent, consisting of data structures and algorithms, capable of processing the operations of the stream online in sequence (without the possibility of returning to already processed operations). At any point, upon external request the agent should be able to return a set of frequent elements. We will say that the output of the algorithm is correct if the returned set contains $O(1/\varphi)$ elements, including all elements with frequency at least φ and no element with frequency smaller than or equal to $\varphi-\epsilon$.

Performance of an agent is measured in terms of time to process a single operation from the stream, time to output all the (φ,ϵ)-hot elements, and the total local memory used. In the measurement, the atomic operation concerns so called *Machine word*: a unit of memory that is sufficient to store a single element, step number and all problem parameters; every basic operation on machine words, e.g., arithmetics, is accounted as 1 in time complexity. We aim at time and memory efficient agents, i.e., performing each operation (or outputting an element) in time $O(\text{ polylog } n)$ and using $O(\frac{1}{\epsilon}\text{ polylog } n)$ memory units (each storing a machine word).

Additional notation. By t denote the current step number (i.e., the number of operations of the stream that have already been handled by the algorithm). We will also use notation $\gamma=\frac{\epsilon}{6}$.[1]

1.1 Previous and Related Work

Handling insertions and deletions. [Cormode and Muthukrishnan, 2005b] proposed randomized online algorithm with memory $O(k\log k\log n)$, processing each operation in time $O(\log k\log m)$, and outputting hot elements in time $O(k\log k\log m)$, where $k=\frac{1}{\varphi}-1$. The algorithms return no items with frequencies less than $\frac{1}{k+1}-\epsilon$ with some user-specified probability. A deterministic summary structure for data streams in [Ganguly and Majumder, 2007] finds the most frequent elements however it requires space $O(\log^3 n/(\varphi^2\varepsilon^2))$. In [Cormode and Muthukrishnan, 2005a], the objective is to return an approximate frequency of any element using memory $\widetilde{O}(1/\varepsilon)$ and time $\widetilde{O}(1)$, but the error was proportional to ε times the total frequency of all the elements. In [Gilbert *et al.*, 2002b] the considered problem is to return approximate quantiles of the data – the solution uses space $O(1/\varepsilon^2)$.

Earlier results include the problem of histogram maintenance, which involves finding a piecewise-constant approximation of data distribution. The optimal histogram is close to the data vector in terms of ℓ_1 or ℓ_2 norms hence it approximates all the data points, whereas in the problem of hot elements the objective is to approximate the frequencies of only the most frequent elements.[Gibbons *et al.*, 1997] were the first who considered insertions and deletions in the context of maintaining various histograms, however their methods need periodical access to large portion of the data in the presence of deletes. [Gibbons and Matias, 1998;

[1] Auxiliary parameters in this work are chosen for convenience of mathematical analysis in our general streaming model, without harming asymptotic performance. Further optimization of constants could be possible through more detail case study and/or specific experimental optimization for selected datasets.

Gibbons and Matias, 1999] analyzed mainly insertion operations, but also performed experimental study in the presence of deletions. [Gilbert *et al.*, 2002a] designed and analyzed algorithms for maintaining histograms with guaranteed accuracy and small space.

Insertion-only streams. For streams with only insertions, in a special case without a lower bound on the frequency of returned elements (which is equivalent to $\epsilon = \varphi$), [Misra and Gries, 1982] designed a deterministic algorithm with processing and enlisting time $O(k\log k)$ and memory $O(k)$ (which in this case equals to $O(1/\epsilon)$). Their algorithm however is not fully online, as it requires a second pass on the stream. In the same model, [Demaine *et al.*, 2002] proposed a single-pass randomized algorithm for finding frequent elements using $O(k)$ memory and $O(1)$ expected time for processing one item.

In the more general case, [Manku and Motwani, 2002] proposed a deterministic algorithm Lossy Counting that processes finds hot elements using $O(\log(n/\epsilon)/\epsilon)$ memory. [Charikar *et al.*, 2002] proposed a randomized algorithm using $O(k\log n/\epsilon^2)$ memory and and $O(\log(1/\sigma))$ time per operation that succeeds with probability at least $1-\sigma$.

Recent works. Recently, results on finding frequent items in streams include models where later items in the stream have higher weight [Wu *et al.*, 2017; Cafaro *et al.*, 2017], as well as many applications, e.g., finding frequent elements in two-dimensional data streams [Lahiri *et al.*, 2016; Epicoco *et al.*, 2018].

2 Algorithm

2.1 Data Structures with Operations

Our agent maintains two data structures: a *disperser-based* structure of *group counters* and a structure of *individual entries* built on the top of *balanced Binary Search Trees*.

Definition of disperser. Formally, a disperser $G=(V,W,E)$ is a bipartite graph satisfying the following criteria:

Left-degree: G has left-degree d (i.e., each vertex in V has d neighbors in W),

Right-set: $|W|=\phi\ell d/\delta$,

Dispersion: for every $L\subseteq V$ such that $|L|\geq\ell$, the set $\Gamma_G(L)$ of neighbors of L in graph G is of size at least $(1-\xi)|W|$.

Construction of disperser. [Ta-Shma *et al.*, 2001] showed a construction of a disperser with construction parameters: left-degree $d=$ polylog n and entropy loss $\delta=O(\log^3 n)$, such that for each $v\in V$ neighborhood $\Gamma_G(v)$ can be enlisted in time and memory polylog n. They also mentioned existence of dispersers with $d,\delta=O(\log n)$.

Group counters. To build this structure, we use a is an (ℓ,d,ξ)*-disperser graph* G *with entropy loss* δ for parameters: $|V|=n$, $|W|=\phi\frac{d}{2\xi\gamma}$ for some sufficiently large constant $\phi>1$, $0<\xi<1/2$ is an arbitrary chosen constant, d,δ depend on the construction of disperser (see the construction comments later on), and $\ell=\frac{\delta}{2\xi\gamma}$ (recall our notation $\gamma=\frac{\epsilon}{6}$). Each element $x\in N$ is associated with a unique vertex $v_x\in V$ and each node $w_g\in W$ is associated with a group counter g, where the group is the set $\Gamma_G(v_x)\subseteq V$ of neighbors of w_g in G. The set of group counters $\mathcal{G}(x)$ of an element $x\in N$ is the set of group counters associated with neighbors of v_x in graph G, i.e., $\mathcal{G}(x)=\{g: w_g\in\Gamma_G(v_x)\}$. We will be using v_x and x interchangeably, whenever it does not raise any confusion; similarly, w_g and g.

Individual entries. To account for operations on elements that need to be counted precisely, we introduce a new structure $\mathcal{C}$ of *individual entries* with supporting procedures. The entries will be kept for two types of elements: candidates (potential hot elements with large number of insertions minus deletions) and recently modified elements (elements for which there exists an operation insert or delete at most $\lceil 2/\gamma\rceil$ steps ago). The reason why we need to keep individual entries for recently modified elements is that we do not process an incoming operation immediately. Instead, upon arrival of an operation our agent inserts it to an auxiliary queue $\mathcal{Q}$ and updates group counters and individual entries (at most) $\lceil 2/\gamma\rceil$ steps later. All these happen in the local memory of the agent (of limited capacity) and does not cause the agent to go backwards the stream nor looking ahead. Individual entry of an element x consists of:

- *key* x of the element $x\in N$;
- *candidate counter* $c(x)$, which is incremented/decremented by 1 upon handling of each insertion/deletion of element x;
- *recent counter* $r(x)$, which stores the number of insertions minus the number of deletions of element x in recent operations in the stream;
- *number of operations* $\lambda(x)$, which stores the number of recent operations on element x.

In each step, our agent receives a single operation from the stream. Note that after the algorithm processes an operation, it cannot go back to it (i.e., its algorithm only makes a single online pass over the stream). In the processing, the following procedures are used on datastructure $\mathcal{C}$:

check(x)**:** it checks if there is an entry with key x. If there is, it returns $\langle c(x),r(x),\lambda(x)\rangle$, otherwise it returns $null$; it takes time $O(\log|\mathcal{C}|)$;

add(x)**:** it adds an entry element with key x and initial values of all counters $c(x)$, $r(x)$, $\lambda(x)$ equal to 0; it takes time $O(\log|\mathcal{C}|)$;

size()**:** it returns the number of entries; works in time $O(1)$;

apply_recent(op,x)**:** it applies operation op on recent counter $r(x)$ of element x; it checks if there is an entry with key x in the balanced tree $\mathcal{T}$; if there is no such entry, it creates it; then it increments $\lambda(x)$ and increments or decrements (depending on the type of op) the recent counter $r(x)$; it takes time $O(\log|\mathcal{C}|)$;

rollback_recent(op,x)**:** it performs rollback of operation op on an entry with key x; it accesses entry with key x; decrements $\lambda(x)$, decrements (if op$=$insert) or increments (if op$=$delete) $r(x)$; it takes time $O(\log|\mathcal{C}|)$;

apply_candidate(op,x)**:** it applies operation op on the candidate counter of entry with key x; it finds a copy of an entry with key x in the balanced tree and increments (if op$=$insert) or decrements (if op$=$delete) its counter $c(x)$ by 1; it takes time $O(\log|\mathcal{C}|)$;

remove_if_small(x,s)**:** it removes element key x if the minimum group counter of x is below threshold s and if the number of recent operations equals to zero (i.e., $\lambda(x)=0$); it takes time $O($ polylog $n)$ (because this is the time to access all the group counters to which v belongs);

get_larger_than(s)**:** it returns a list of keys for which the corresponding candidate counter plus recent counter exceeds s; it accesses the entries in order (the structure is sorted by $c(x)+r(x)$) and returns the elements as long as the value is greater than s. It takes time proportional to the number of returned elements.

Implementation of datastructure $\mathcal{C}$. Implementation of datastructure $\mathcal{C}$ with time complexities of individual operations, as claimed in the previous paragraph, can be achieved using standard datastructures. We use a balanced Binary Search Tree T_1 (e.g., Red-Black Tree, c.f., [Aho *et al.*, 1983]), with keys being the identifiers of all the elements for which an individual entry exists and entries being tuples of a type $\langle r(x), c(x), \lambda(x), p_x\rangle$. This ensures that searching and returning the individual entry of an element (if it exists) takes logarithmic time. Each entry of T_1 has a pointer p_x to an entry in a second Balanced Binary Search Tree T_2 with keys being the values of $r(x)+c(x)$ and entries being tuples $\langle x,q_x\rangle$, where q_x points towards the entry of element x in tree T_1. Clearly, any update of counters $r(x), c(x), \lambda(x)$ for some element x requires logarithmic time operation to find the entry in T_1 and then logarithmic time to modify the key of a corresponding entry in T_2. Inserting and deleting elements also requires logarithmic time. Listing the elements (operation get_larger_than(s)) quickly is feasible since the inorder traversal of tree T_2 returns the elements in order of decreasing $r(x)+c(x)$. The time of the operation is proportional to the number of returned elements, multiplied by $O(\log n)$.

2.2 Main Algorithm

Intuitions. There is a following interplay between the two data structures (group counters and individual entries) in the agent's algorithm. The group counters determine which elements should be included in the individual entries as candidates for hot elements. The individual entries keep track of the operations on the candidates for hot elements and determine which elements should be returned as hot.

Processing of an operation. In our algorithm, each element $x \in N$ has an associated set of group counters $\mathcal{G}(x)$ (note that each group counter is shared by multiple elements). Processing of operation op(x) (where op is either insert or delete) involves updating (incrementing or decrementing, resp.) each of the counters $\mathcal{G}(x)$. If all the group counters $\mathcal{G}(x)$ are above a threshold $\gamma \cdot t$, it indicates that the element may be a candidate for being a hot element. In this case, the agent checks the structure of individual entries $\mathcal{C}$ and updates the candidate counter $c(x)$ in the entry of x, or creates it if it does not exist. In our algorithm we define the *candidate* as an element with at least $\gamma \cdot t$ net occurrences (insertions minus deletions) in the first t operations of the stream.

Group counters. A group counter g is incremented when any element from the group (recall that the group is determined by the topology of the used disperser G) is inserted in the stream. This means that some element x might have all its group counters $\mathcal{G}(x)$ above $\gamma \cdot t$ while its net occurrence $n_t(x)$ could be below $\gamma \cdot t$ (because insertions of other elements have caused the counters in $\mathcal{G}(x)$ to exceed the $\gamma \cdot t$ threshold). Hence, some 'false positive' candidates might be included in $\mathcal{C}$. In our analysis we will bound the number of such false positives. We ensure this by proving that in any set of size $\Theta(\delta/\gamma)$,some element will have a group counter with value at most $\gamma \cdot t$. (Recall δ is the disperser's entropy loss.)

Algorithm 1: Handling $2\lceil \frac{1}{\gamma} \rceil$ consecutive operations.

```
t ← current step number;
γ ← ϵ/6, τ ← ⌈1/γ⌉;
// Phase 1
Divide elements included in C into τ chunks
  E_1,...,E_τ each of (at most) ⌈C.size()/τ⌉ elements;
for i ← 1 to τ do
    Receive the next operation op(x);
    Q.enqueue(op(x));
    C.apply_recent(op, x);
    foreach v ∈ E_i do
        C.remove_if_small(v, ⌈γt⌉);
// Phase 2
for i ← 1 to τ do
    Receive the next operation op(x);
    Q.enqueue(op(x));
    C.apply_recent(op,x);
    // Process two items from the queue
    Process(Q.dequeue(), C, γ, t+2(i−1));
    Process(Q.dequeue(), C, γ, t+2(i−1)+1);
```

Algorithm 2: Extracting hot elements at step t.

```
return C.get_larger_than((φ−ϵ)t);
```

Procedure Process(op(x), $\mathcal{C}$,γ, t)

```
// Processing a single operation
Compute set G(x) of group counters of x;
foreach g ∈ G(x) do
    if op = insert then g ← g+1 ;
    else g ← g−1;
C.rollback_recent(op, x);
if C.check(x) then
    C.apply_candidate(op, x);
else
    g_min ← min{g : g ∈ G(x)};
    if g_min ≥ ⌈γt⌉ then
        C.add(x);
        C.apply_candidate(op, x);
```

Candidates. The key property guarantying that our algorithm is correct is the following: in every step t, each element with at least $\gamma \cdot t$ net occurrences has its individual entry in $\mathcal{C}$. It is nontrivial to show, as this property must hold in every step, and the set of such elements might change over time. We have to make sure that when an element exceeds the threshold, an individual entry for this element must be created immediately. We ensure this by (1) keeping individual entries for all elements for which there is even a single recent operation, and (2) inserting an element into $\mathcal{C}$ when each of its group counters exceeds $\gamma \cdot t$. With these properties, we know that each hot element is at any time in $\mathcal{C}$ and, moreover, at the moment such an element is inserted into $\mathcal{C}$ it is just crossing the $\gamma \cdot t$ threshold. Because the occurrences are counted precisely after an element is added to $\mathcal{C}$, we miss only $\gamma \cdot t_x$ net occurrences of an element x inserted into $\mathcal{C}$ at step t_x. Since $\gamma < \epsilon$ and $t_x \leq t$, each hot element has at least $(\varphi - \epsilon) \cdot t$

net occurrences while having an individual entry. This allows our algorithm to correctly find all the hot elements.

Pseudo-codes. The algorithm is described on three pseudo-codes: one handling $2\lceil 1/\gamma \rceil$ consecutive operations in the stream, c.f., a pseudo-code in Algorithm 1, a sub procedure that processes a single operation, c.f., procedure Process, and the final pseudo-code enlisting hot elements in the processed stream, c.f., a pseudo-code in Algorithm 2. Note that a request to return hot elements can arrive at any step (also while handling $2\lceil 1/\gamma \rceil$ consecutive operations). In such case, the execution of Algorithm 1 is paused and Algorithm 2 is immediately executed.

The latter is a single execution of procedure get_larger_than(s) for $s=(\varphi-\epsilon)t$, where t is the current position in the stream.

The former, Algorithm 1, works as follows. In a single iteration of the algorithm, processes $2\lceil 1/\gamma \rceil$ consecutive operations, online one-by-one. This part of the stream, also called a window (recall though the agent does not process whole window at once, but online one-by-one), is further divided into two sub-windows of $\lceil 1/\gamma \rceil$ operations each. In the first sub-window the agent performs Phase 1, during which it executes the cleanup of $\mathcal{C}$ (see Algorithm 1 lines 4-9). During these steps, the new operations are pre-processed (using apply_recent method of $\mathcal{C}$) and inserted into the queue $\mathcal{Q}$. In each step of processing the second sub-window (Phase 2), the agent finishes processing of some pair of operations from the queue (see Algorithm 1 lines 10-15). Hence, the length of the queue decreases by 1 in each step of Phase 2, and thus after the last step the queue is empty. Note that our algorithm, even though it processes operations in windows, is strictly online, because in each step the agent receives a new operation from the stream and each received operation is immediately handled.

3 Analysis of the Algorithm

In the analysis, for the sake of clarity, we sometimes add lower index t to some model variables to indicate that the values of these variables are taken just after processing t operations from the stream, e.g., $\mathcal{G}_t(x)$ denotes the set of all group counters of element x after processing the first t operations.

The first lemma shows that there is a limited number of elements $x \in N$, namely $\frac{\delta}{2\xi\gamma}$, having all its group counter at least $\lceil \gamma t \rceil$. The second lemma shows that the number of recent operations stored in the recent counters is at most $\lceil 1/\gamma \rceil + 1$. Thus procedure remove_if_small executed on all the elements from $\mathcal{C}$ removes all but at most $\frac{\delta}{2\xi\gamma} + \lceil 1/\gamma \rceil + 1$ entries. When an operation arrives we first *pre-process* it by inserting into queue $\mathcal{Q}$ (and upon insertion, this operation is accounted for in a recent counter). Some number of steps later we *finish processing* the operation, by executing procedure $Process$ with this operation as the first argument.

Lemma 1. *After processing first t stream operations by Algorithm 1, for any set $S \subseteq N$, where $|S| = \frac{\delta}{2\xi\gamma}$, there is an element $s \in S$, such that $g < \gamma t$ for some of its group counters $g \in \mathcal{G}_t(s)$.*

Proof. First observe, that since in upon processing an operation, we increment at most $d = \operatorname{polylog} n$ group counters (recall that d is the left-degree of the disperser G used for the construction, c.f., [Ta-Shma *et al.*, 2001]), then after processing t operations from the stream the sum of the group counters is at most td. Since there are $|W| = \frac{\phi d}{2\xi\gamma}$ group counters (where a constant $\phi > 1$), associated 1-1 with the right vertices of disperser G, then the average value of a group counter is at most $td/|W|$. At least 2ξ-fraction of the counters must be below $1/(2\xi)$ times the average; thus, there are at least $2\xi|W|$ group counters with values smaller than $td/(2\xi|W|) \leq \gamma t$, where $\xi < 1/2$ is a parameter of the disperser G and W of size $|W| \geq \frac{d}{2\xi\gamma}$ is the number of its right-side vertices. If we take any set $S \subseteq N$ of size at least $\frac{\delta}{2\xi\gamma}$,the set $N_G(S)$ of neighbors of S in graph G is of size at least $(1-\xi)|W|$, by dispersion. Hence at least $2\xi|W| + (1-\xi)|W| - |W| = \xi|W|$ counters associated with some elements in S must be below γt. □

Lemma 2. *At any step of the algorithm, the total number of operation stored in recent counters is at most $\tau+1$.*

Proof. It is easy to see that the size of queue $\mathcal{Q}$ in Algorithm 1 is always at most $\tau+1$. It is because it is empty at the beginning of the algorithm and then we add a single operation to it in each iteration of for-loop in lines 4-9 of Algorithm 1 and then in each iteration of the for-loop in lines 10-15 of Algorithm 1 we first add a single operation and then remove two. Since both for-loops have τ iterations each, the maximum size of the queue is after line 12 during the first iteration of the second for-loop and equals to $\tau+1$. Hence the total number of operations stored in the recent counters is also at most $\tau+1$. It remains to observe that each operation in the queue contributes exactly one to the sum $\sum_{v \in N} \lambda(v)$. □

Lemmas 1 and 2 imply an upper bound on the size of structure $\mathcal{C}$.

Lemma 3. *1. After each Phase 1, the number of individual entries in $\mathcal{C}$ is at most $\frac{\delta}{2\xi\gamma} + \lceil 1/\gamma \rceil + 1$;*

2. After each Phase 2, the number of individual entries in $\mathcal{C}$ is at most $\frac{\delta}{2\xi\gamma} + 3\lceil 1/\gamma \rceil + 1$.

Next we relate the number of net occurrences $n_t(x)$ of an element $x \in N$ in the stream by position t, to counters $c(x)$ and $r(x)$.

Lemma 4. *After first t steps of Algorithm 1:*

1. for every element $x \in N$, if $n_t(x) \geq \lceil \gamma t \rceil$, then $x \in \mathcal{C}_t$,

2. for every element $x \in \mathcal{C}_t$ we have $n_t(x) - \lceil \gamma t \rceil \leq c_t(x) + r_t(x) \leq n_t(x)$.

Proof. To show (1), we need to show, that in the algorithm after processing t' operations all elements with at least $\lceil t'\gamma \rceil$ occurrences are in $\mathcal{C}$. To prove this we need to show that we never remove from $\mathcal{C}$ an element with at least $\lceil t'\gamma \rceil$ net occurrences and that we always add to $\mathcal{C}$ an element when its number of net occurrences becomes equal to or larger than $\lceil t'\gamma \rceil$. We prove this fact by induction. Assume that it holds for all phases up to some step t'. We then consider the following Phase 1 and Phase 2. Note that the queue $\mathcal{Q}$ is empty at the beginning of each Phase 1. It is because each Phase 1 adds τ operations to it and each Phase 2 removes τ operations from the queue. Note that, when the queue is empty, each group counter contains the total number of insertions minus number of deletions of all the elements from the group (because we processed all the operations up to this step). This means that any element removed from $\mathcal{C}$ by operation remove_if_small has some group counter at most $\lceil t\gamma \rceil$, hence it surely has at most $\lceil t\gamma \rceil$ occurrences in the first t operations in the stream. Additionally, we keep in $\mathcal{C}$ individual entry for

each element with at least one (recent) operation in the interval $[t,t+\tau]$. Hence after Phase 1 we have in $\mathcal{C}$ all the elements that have at least $\lceil t\gamma \rceil$ occurrences in the first t operation or at least one operation in the interval $[t,t+\tau]$. Hence after Phase 1 we have that $\mathcal{C}$ contains all the elements that have at least $\lceil (t+\tau)\gamma \rceil$ occurrences. In Phase 2 we add elements to $\mathcal{C}$ only upon handling of some t'-th operation $\mathsf{op}(x)$ if each group counter of x equals to at least $\lceil t'\gamma \rceil$. Hence, after handling t'-th operation in Phase 2, the claim holds. This finishes the inductive proof.

To show (2), we first observe that if an element $x \in \mathcal{C}$, then upon processing an opration on x, this operation is immediately accounted for in counter $r(x)$. Hence $r(x)$ is incremented/decremented upon each insertion/deletion of element x as long as x belongs to $\mathcal{C}$. Note that upon handling of an operation, the sum $r(x) + c(x)$ does not change. Hence $r_t(x)+c_t(x)$ stores the number insertions minus the number of deletions of element x since the step when x was inserted into $\mathcal{C}$. This means that $c_t(x)+r_t(x) \le n_t(x)$. To show that left side of (2), consider the largest time step $t' < t$, such that at step t', $x \notin \mathcal{C}$. We know from (1), that $n_{t'}(x) < \lceil \gamma t' \rceil$. We also know that as long as x belongs to $\mathcal{C}$, $c_t(x)+r_t(x)$ counts all the occurences of x, hence $c_t(x)+r_t(x) = n_t(x)-n_{t'+1}(x) \ge n_t(x)-\lceil \gamma t' \rceil$. □

Theorem 1. *Algorithms 1 and 2 correctly solve the problem of deterministically finding (φ,ϵ)-hot elements using $O(\log^3 n/\epsilon)$ memory, $O(\text{ polylog } n)$ worst-case time for processing a single operation, and $O(\log n/\varphi)$ time for enlisting the (φ,ϵ)-hot elements.*

Proof. To show that the main algorithm, consisting of Algorithms 1 and 2, is correct we need to show two facts:

1. it returns all elements with at least $t\varphi$ occurrences,
2. it returns no element with less than $t(\varphi-\epsilon)$ occurrences.

The second fact follows directly from Lemma 4(2), because if $t(\varphi-\epsilon) > n_t(x)$, then also $t(\varphi-\epsilon) > c(x)+r(x)$, and such element is not returned by procedure get_larger_than.

The first part follows from Lemma 4(1). If $t \le 1/\gamma$, then all the insertions minus deletions are stored in the recent counters (i.e., $r_t(x) = n_t(x)$), hence if $n_t(x) \ge t\varphi$, then $r(x) \ge t(\varphi-\epsilon)$ and this element will be returned by procedure get_larger_than().

Otherwise, if $t > 1/\gamma$ if we have an element x, with $n_t(x) > t\varphi = t(\varphi-\epsilon) + 6\gamma t > t(\varphi-\epsilon) + t\gamma + 5 > t(\varphi-\epsilon) = t(\varphi-\epsilon) + \lceil t\gamma \rceil$. Then by Lemma 4(1) $x \in \mathcal{C}_c$ and $c_t(x)+r_t(x) \ge n_t(x)-\lceil \gamma t \rceil \ge t(\varphi-\epsilon)$ and this element will be returned by procedure get_larger_than().

The memory complexity of our algorithm is straighforward from the definition of the data structures (the dominating part is the structure of group counters, and the overhead $\log^3 n$ comes from the value of δ in the constructive disperser). The time of enlisting the hot elements is in fact: $O(1/(\varphi-\epsilon))$ elements times $O(\log n)$ per element (see the definition of procedure get_larger_than()), which clearly gives $O(\log n/\varphi)$ for $\varphi - \epsilon = \Theta(\varphi)$; in case $1/(\varphi-\epsilon) = o(1/\varphi)$ we can run our algorithm for finding $(\varphi,\varphi/2)$-hot elements rather than original parameters φ,ϵ, which clearly returns only valid (φ,ϵ)-hot elements. Polylogarithmic time per operation follows from the polylogarithmic time of enlisting group counters of a single element (using disperser) and a constant number of calls to logarithmic-time procedures on the individual entries structure. □

4 Lower Bound on Memory

Theorem 2. *Any deterministic algorithm finding (φ,ϵ)-hot elements requires memory $\Omega(\frac{1}{\epsilon})$, even for insertions.*

Proof. Let $\alpha = \lceil \frac{1}{\varphi-\epsilon} \rceil - 1$. W.l.o.g. assume $\alpha \in Z$. Consider any deterministic algorithm finding (φ,ϵ)-hot elements. We design a stream to enforce the lower bound $\Omega(\frac{1}{\epsilon})$ on its memory size.

Let X be an arbitrary set of $x = \lfloor \frac{1-\varphi+\epsilon}{\epsilon} \rfloor$ elements. Note that $x = \Theta(1/\epsilon)$. First part of the stream $\mathcal{S}(X)$ associated with set X contains α insertions of each element from set X, starting from α insertions of the first element in X, then α insertions of the second element, etc. Next, the adversary adds x insertions of element $v \in N$ to the previously constructed stream and checks whether element v is hot after these operations. Observe that if $v \notin X$, we have $f(v) = \frac{x}{\alpha x+x} = \frac{1}{\alpha+1} \le \varphi-\epsilon$. Otherwise, i.e., if $v \in X$, we get

$$f(v) = \frac{\alpha+x}{\alpha x+x} = \frac{\alpha/x+1}{\alpha+1} \ge \frac{\frac{(1/(\varphi-\epsilon))-1)\cdot\epsilon}{1-\varphi+\epsilon}+1}{1/(\varphi-\epsilon)} = \frac{\frac{\epsilon}{\varphi-\epsilon}+1}{1/(\varphi-\epsilon)} = \varphi\,.$$

Since there are $\binom{n}{x}$ choices of set X, the number of bits to uniquely encode all possible sets X is $\log_2 \binom{n}{x} = \Theta((1-(\varphi-\epsilon))/\epsilon \cdot \log n)$, which is $\Theta(1/\varepsilon)$ machine words. Consider an algorithm $\mathcal{A}$ for finding (φ,ϵ)-hot elements using less memory. Since the memory size is smaller than $\log_2 \binom{n}{x}$ machine words, there exist two sets $X_1 \neq X_2$, $|X_1|,|X_2| = x$, such that after processing stream $\mathcal{S}(X_1)$ the memory state is exactly the same as after processing stream $\mathcal{S}(X_2)$. We showed that after processing $\mathcal{S}(X_1)$ all and only elements from X_1 can be returned by the algorithm as hot element. On the other hand, in $\mathcal{S}(X_2)$, all and only elements from X_2 can be returned by the algorithm as hot element. This is a contradiction with the assumption that $\mathcal{A}$ is a correct deterministic algorithm for finding (φ,ϵ)-hot elements. □

5 Extensions, Discussion and Open Directions

False positives/negatives and stochastic counterpart. Our approach guarantees no false-negatives, one could turn it into no false-positives by running the agent for parameters $\varphi+\epsilon,\epsilon$ instead of φ,ϵ, resp. If the stream is stochastic, our agent returns *all and only* elements with frequency at least φ with probability corresponding to the distribution of hot items, e.g., for power-law distribution the probability of false positive/negative is polynomial in ϵ.

Graphs, hypergraphs, relationships and weights. Our agent could be applied if the universe N contains a relationship $\mathcal{R}$ on some set $\mathcal{N}$, e.g., a graph or a hypergraph. Our agent not only monitors high frequencies of (hyper-)edges, but also vertices from set $\mathcal{N}$ or sets in any other sub-relation $\mathcal{R}'$ of $\mathcal{R}$, with a slight performance overhead $\max_{R \in \mathcal{R}} |\{R' : R' \subseteq R \ \ \& \ R' \in \mathcal{R}'\}|$. Moreover, instead of inserting or deleting elements, we could implement more complex operations of updating variables in set N.

Parallel version. When operations appear faster than a single agent can handle, we can employ multiple agents to process the operations. Parallelization of our algorithm is possible assuming that the agents have parallel access to queue $\mathcal{Q}$, structure $\mathcal{C}$ and the structure of group counters. When having κ agents, for $\kappa < \tau$, we can parallelize the loops in lines 4 - 9 and 10 - 15 of Alg. 1. We handle κ operations in each parallel step (and $\kappa \bmod \tau$ once every $\lceil \tau/\kappa \rceil$ parallel step) and we retain the same memory complexity of the algorithm (i.e., agents could use a small shared memory).

Acknowledgments

Partially supported by Polish National Science Center (NCN) grant 2019/33/B/ST6/02988 and by the NSF grant 2131538.

References

[Aho *et al.*, 1983] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1st edition, January 1983.

[Alon *et al.*, 1996] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *STOC 1996*, pages 20–29. ACM, 1996.

[Cafaro *et al.*, 2017] Massimo Cafaro, Italo Epicoco, Marco Pulimeno, and Giovanni Aloisio. On frequency estimation and detection of frequent items in time faded streams. *IEEE Access*, 5:24078–24093, 2017.

[Charikar *et al.*, 2002] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *ICALP 2002*, pages 693–703. Springer, 2002.

[Cormode and Muthukrishnan, 2005a] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[Cormode and Muthukrishnan, 2005b] Graham Cormode and Shan Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)*, 30(1):249–278, 2005.

[Demaine *et al.*, 2002] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 348–360. Springer, 2002.

[Epicoco *et al.*, 2018] Italo Epicoco, Massimo Cafaro, and Marco Pulimeno. Fast and accurate mining of correlated heavy hitters. *Data Mining and Knowledge Discovery*, 32(1):162–186, 2018.

[Fang *et al.*, 1998] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *VLDB 1998*, pages 299–310. Morgan Kaufmann, 1998.

[Ganguly and Majumder, 2007] Sumit Ganguly and Anirban Majumder. Cr-precis: A deterministic summary structure for update data streams. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium, ESCAPE 2007, Hangzhou, China, April 7-9, 2007, Revised Selected Papers*, volume 4614 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2007.

[Gibbons and Matias, 1998] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 331–342. ACM Press, 1998.

[Gibbons and Matias, 1999] Phillip B. Gibbons and Yossi Matias. Synopsis data structures for massive data sets. In *SODA*, pages 909–910. ACM/SIAM, 1999.

[Gibbons *et al.*, 1997] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *VLDB 1997*, pages 466–475. Morgan Kaufmann, 1997.

[Gilbert *et al.*, 2002a] Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *STOC 2002*, pages 389–398. ACM, 2002.

[Gilbert *et al.*, 2002b] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB 2002*, pages 454–465. Morgan Kaufmann, 2002.

[Ioannidis and Christodoulakis, 1993] Yannis E. Ioannidis and Stavros Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.*, 18(4):709–748, 1993.

[Ioannidis and Poosala, 1995] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 233–244. ACM Press, 1995.

[Karp *et al.*, 2003] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[Kushilevitz and Nisan, 1997] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.

[Lahiri *et al.*, 2016] Bibudh Lahiri, Arko Provo Mukherjee, and Srikanta Tirthapura. Identifying correlated heavy-hitters in a two-dimensional data stream. *Data mining and knowledge discovery*, 30(4):797–818, 2016.

[Lin *et al.*, 2005] Chih-Hsiang Lin, Ding-Ying Chiu, Yi-Hung Wu, and Arbee LP Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 68–79. SIAM, 2005.

[Manku and Motwani, 2002] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB 2002*, pages 346–357. Morgan Kaufmann, 2002.

[Misra and Gries, 1982] Jayadev Misra and David Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[Ta-Shma *et al.*, 2001] Amnon Ta-Shma, Christopher Umans, and David Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In *STOC 2001*, pages 143–152. ACM, 2001.

[Wu *et al.*, 2017] Shanshan Wu, Huaizhong Lin, Yunjun Gao, Dongming Lu, et al. Novel structures for counting frequent items in time decayed streams. *World Wide Web*, 20(5):1111–1133, 2017.