Data Stream Algorithms I

Joaquim Madeira

Version 0.2 – November 2018

Overview

- The data stream model
- Finding frequent items
- The MAJORITY problem
- The FREQUENT problem

Data Streams

- Many data generation processes can be modeled as data streams
  - Huge numbers of simple pieces of data
  - Arriving at enormous rates
  - Taken together lead to a complex whole
- Hundreds of gigabytes per day or higher !

Data Streams

- Sequence of queries posed to an Internet search engine
- Collection of transactions across all branches of a supermarket chain
- Sequence of packets in network traffic monitoring
- …

Data Streams

- Such data may be archived and indexed within a data warehouse
- BUT, it may also be important to process it "as it happens"
- Up to the minute analysis and statistics on current trends

Data Streams

- Quick response to each new piece of information

- Resources used very small when compared to the total quantity of data

The streaming model

- Data arrives in a streaming fashion
  - Scan the sequence in the given order
  - No random access to the data tokens !
- Must be processed on the fly !
- Accurate computations

The streaming model

- Compute some function $\Phi(\sigma)$ of a massively long input stream σ
- Make just one pass over σ !
- Goal:
  - Use resources (space and time) sublinear on the size of the input !

The streaming model

- When to produce output ?

- At the end of the stream
- When queried on the stream prefix observed so far
- Whenever there is a stream update
- On a "sliding window" of the most recent updates

The basic streaming model

- The data stream:

$$\sigma = \langle a_1, a_2, \dots, a_m \rangle$$

- Each data token a_i is drawn from a set of n elements

- Goal:
  - Process σ using a small amount of memory *s*
  - I.e., make *s* much smaller than *m* and *n* !

The quality of an algorithm's answer

- $\Phi(\sigma)$ is usually a real-valued function
- Allow for
  - Computing an estimate or approximation of $\Phi(\sigma)$
  - Possibly using randomized algorithms
    - That may err with a small, but controllable probability
- How to evaluate the quality of the result ?

The quality of an algorithm's answer

- $A(\sigma)$ is the output of a randomized algorithm
  - It is a random variable !
- (ε, δ)-approximation of $\Phi(\sigma)$

$$P\left(\left|\frac{A(\sigma)}{\Phi(\sigma)} - 1\right| > \varepsilon\right) \leq \delta$$

- (ε, δ)-additive-approximation of $\Phi(\sigma)$

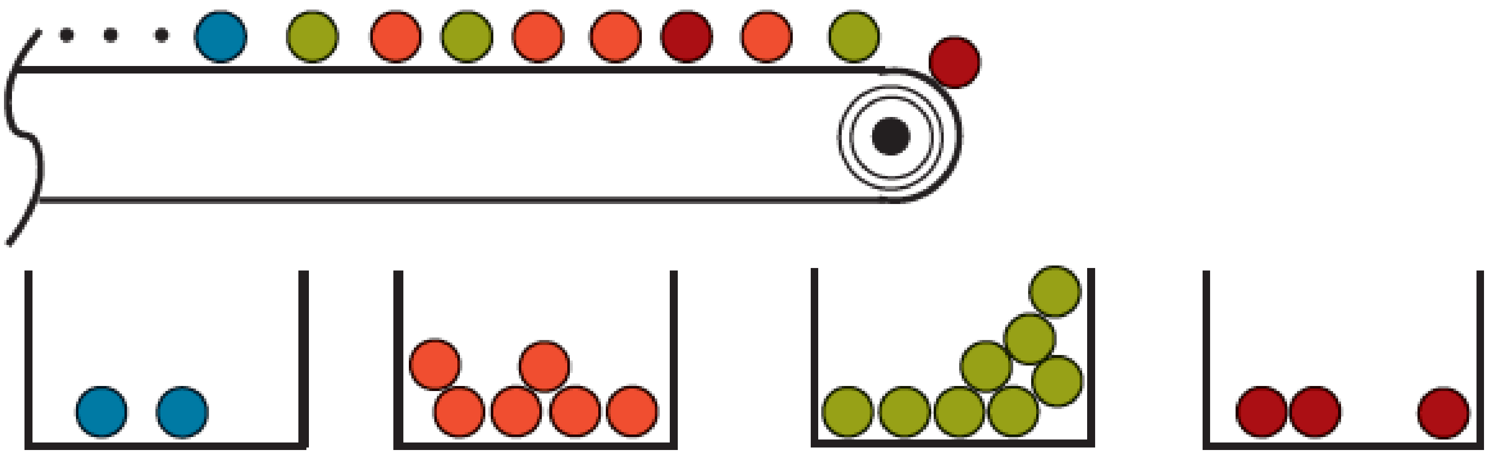
$$P(|A(\sigma) - \Phi(\sigma)| > \varepsilon) \leq \delta$$

Finding frequent items

- The frequent items / “heavy-hitters” problem
- Given a sequence of items, identify those which occur most frequently
- More formally :
- Find all items whose frequency exceeds a specified fraction of the total number of items

Finding frequent items

Figure 1. A stream of items defines a frequency distribution over items. In this example, with a threshold of $\phi = 20\%$ over the 19 items grouped in bins, the problem is to find all items with frequency at least 3.8—in this case, the green and red items (middle two bins).



[Cormode and Hadjieleftheriou]

Finding frequent items

- Network packet monitoring
  - Frequent items represent the heaviest bandwidth users
- Queries made to a search engine
  - Frequent items are the currently popular terms
- …

Finding frequent items

- Counter-based algorithms
  - Track and maintain counts associated with a (varying) subset of stream items
- Sketch algorithms
  - Randomized approach
  - Do not explicitely store stream elements
- Other approaches

Finding frequent items

- Given a stream: $\sigma = \langle a_1, a_2, \dots, a_m \rangle$
- It induces a frequency vector:

$$\boldsymbol{f} = (f_1, f_2, \dots, f_n)$$

$$f_1 + f_2 + \cdots, +f_n = m$$

- The MAJORITY problem
  - If $\exists\, j : f_j > \frac{m}{2}$, then output *j*, otherwise output null
- The FREQUENT problem, with parameter k

The MAJORITY problem

- Applications ?
- Elections
- Fault-tolerant computing
  - Perform multiple redundant computations
  - Check if a majority of the results agree
- …

The MAJORITY problem

- Naïve algorithm for a non-sorted list of values

- Sort the list
- If there is a majority value, it is now the middle value
  - Odd vs even number of list elements

- O(n log n)

- BUT, not useful for data streams !

The MAJORITY problem

- Naïve algorithm
- n frequency counters
- Three-step algorithm
  - Scan the sequence and increment the counters
  - Scan the counters and find the most frequent element
  - Check if it is the majority element : > (m / 2)
- Efficiency ?

The MAJORITY problem

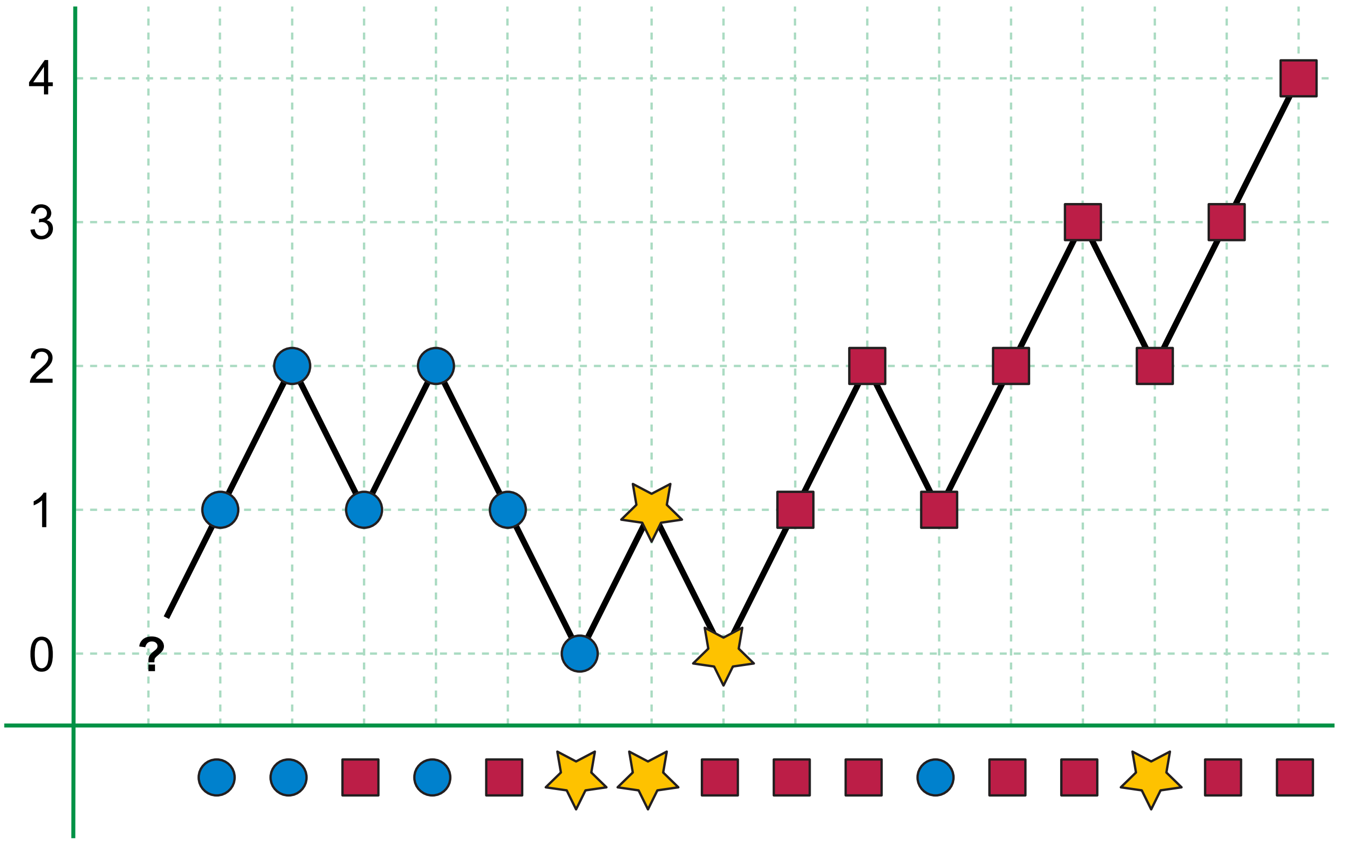
- Boyer & Moore : A fast majority vote alg.
  - 1980
  - http://www.cs.utexas.edu/~moore/best-ideas/mjrty/
- Provided there is such an element, it decides which sequence element is in the majority
- Two-pass algorithm
  - Scan the sequence to identify the majority candidate
  - Scan, again, the sequence, to verify if that candidate is indeed in the majority

MJRTY – A fast majority vote alg.

```
// Initialization
candidate = null; counter = 0;

// First-pass
while ( not end of sequence )
        x = current_token();
        if ( counter == 0 )
        then candidate = x; counter = 1;
        else    if ( candidate == x )
                then counter++;
                else counter--;
```

MJRTY – A fast majority vote alg.



[Wikipedia]

MJRTY – A fast majority vote alg.

```
// Second-pass
counter = 0;
while ( not end of sequence  and counter < (1 + m / 2) )
        x = current_token();
        if ( candidate == x )
        then counter++;
```

- Efficiency ?

MJRTY – A fast majority vote alg.

- Can we skip the second pass?
  - Find a counter-example !

- O(1) extra space
- O(n) time

- BUT, we cannot perform a second pass over a data stream…
  - However, we have a “partial guarantee”

Tasks – The MAJORITY problem

- Implement the naïve algorithm
- Implement the Boyer & Moore algorithm

- Compare their results and running times
  - For random strings over a given alphabet

- For the B & M algorithm, check how many times the majority candidate was indeed the majority

The FREQUENT problem

- The FREQUENT problem, with parameter k
  - Output the set $\{\, j : f_j > m/k \,\}$
- It solves the MAJORITY problem !
- Similar naïve algorithm !
- Can we do better ?

Frequency estimation

- The FREQUENCY-ESTIMATION problem
  - Process the stream σ
  - Establish an estimate for the frequency of any stream token
- Misra & Gries : Finding repeated elements
  - 1982
  - http://www.sciencedirect.com/science/article/pii/0167642382900120
- One-pass algorithm

The Misra & Gries algorithm

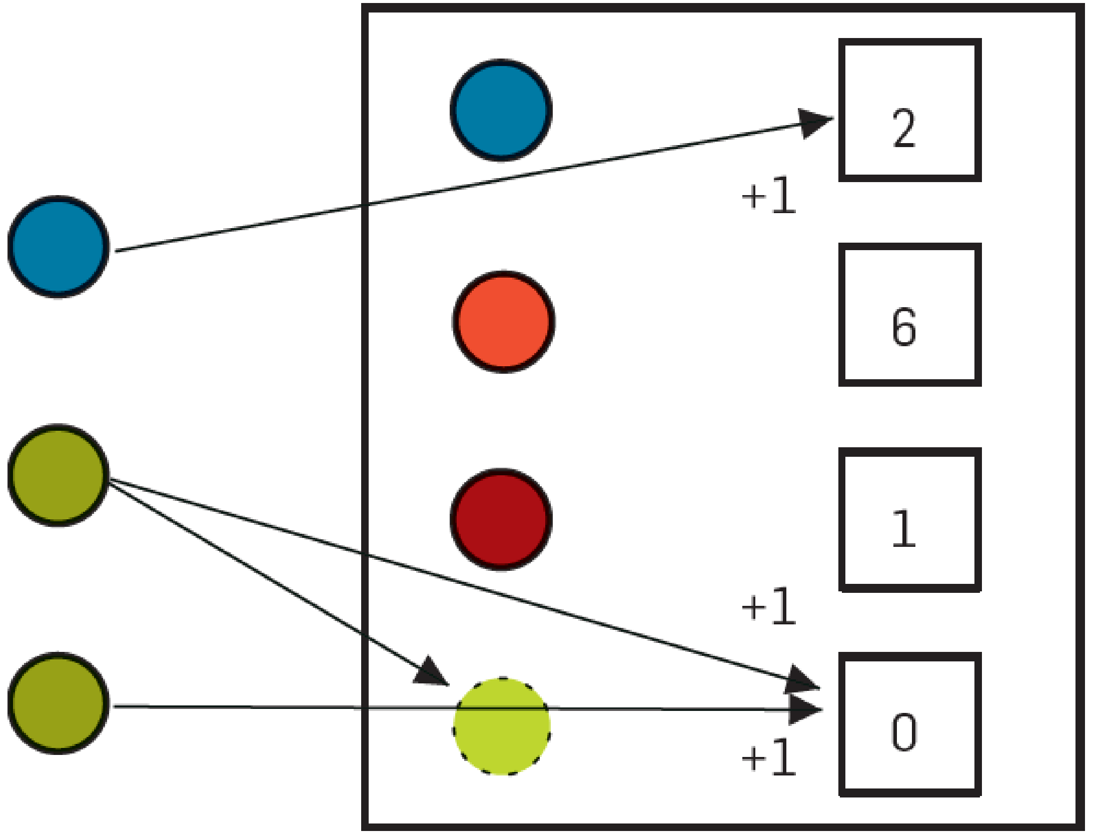
- Parameter k controls the quality of the results given

- It maintains an associative array
  - The keys are tokens seen in the stream
  - Array values are counters associated with the keys / tokens

- At most (k – 1) counters, at any time

The Misra & Gries algorithm

Figure 2. Counter-based data structure: the blue (top) item is already stored, so its count is incremented when it is seen. The green (middle) item takes up an unused counter, then a second occurrence increments it.



[Cormode and Hadjieleftheriou]

The Misra & Gries algorithm

Algorithm 1: FREQUENT(k)

```
n ← 0;
T ← ∅;
foreach i do
    n ← n + 1;
    if i ∈ T then
        c_i ← c_i + 1;
    else if |T| < k−1 then
        T ← T ∪ {i};
        c_i ← 1;
    else forall j ∈ T do
        c_j ← c_j − 1;
        if c_j = 0 then T ← T \ {j};
```

[Cormode and Hadjieleftheriou]

The Misra & Gries algorithm

```
// Initialization
A = empty associative array;
// Processing
while ( not end of sequence )
        j = current_token();
        if ( j in keys(A) ) then A[ j ] = A[ j ] + 1;
        else    if ( | keys(A) | < ( k – 1 ) ) then A[ j ] = 1;
                else    for each i in keys(A) do
                             A[ i ] = A[ i ] – 1;
                             if ( A[ i ] == 0 ) then remove i from A;
// Output
if( a in keys(A) ) then freq_estimate = A[ a ];
else freq_estimate = 0;
```

The Misra & Gries algorithm

- The algorithm, with parameter k, provides, for any token *j*, a freq. estimate f_j^* satisfying

$$f_j - \frac{m}{k} \leq f_j^* \leq f_j$$

  - If some token has $f_j > m / k$, its counter A[j] will be positive

- With an additional pass through the stream, to count the exact frequencies, we can now solve the FREQUENT problem !

Tasks – The Misra & Gries algorithm

- Implement the naïve algorithm
- Implement the Misra & Gries algorithm
  - Choose an appropriate data structure for the associative array

- Test them, using the same input sequences as for the B & M algorithm
  - For different k values !!

Misra & Gries – Recap

- Finds up to (k – 1) items that occur more than a 1/k fraction of the time in the input
- Keeps, at most, (k – 1) candidates at the same time
- No item with frequency m / k is missed
- Algorithm “rediscovered” twice in 2002 !

Implementation issues

- Basic steps
  - Lookup for an item
  - Update a counter
  - Decrement all counters
  - Delete an item with zero counts

- How to ?
  - Optimize speed and space

Implementation issues – Lookup

- Which dictionary data structure ?
- Misra & Gries used a balanced search tree
  - Worst and average case are O(log k)
- Hash table : hash to O(k) buckets
  - Collisions / deletions : how to handle ?
  - Use chaining ?
  - Optimizations ?
- Other ?

Implementation issues – Decrement

- Iterate through all counters : O(k)
- BUT, it happens O(n/k) times
- Optimize ?

- Use a linked list of lists to keep elements grouped by their frequency counts
- Memory space overhead
  - Circular linked lists
  - Also, pointers to and from hash table

Additional algorithms

- There are other algorithms which can be regarded as variations of Misra & Gries' algorithm :

- Lossy-Counting
  - Manku and Motwani, 2002

- Space-Saving
  - Metwally et al., 2005

The Manku & Motwani algorithm

Algorithm 2: LossyCounting(k)

$n \leftarrow 0$; $\Delta \leftarrow 0$; $T \leftarrow \emptyset$;
foreach i **do**
  $n \leftarrow n + 1$;
  if $i \in T$ **then** $c_i \leftarrow c_i + 1$;
  else
    $T \leftarrow T \cup \{i\}$;
    $c_i \leftarrow 1 + \Delta$;
if $\lfloor n/K \rfloor \neq \Delta$ **then**
  $\Delta \leftarrow \lfloor n/k \rfloor$;
  forall $j \in T$ **do**
    if $c_j < \Delta$ **then** $T \leftarrow T \setminus \{j\}$;

[Cormode and Hadjieleftheriou]

Manku & Motwani – Lossy-Counting

- Keep item names and counts
  - Counter value is a lower bound – initially zero
- And an "implicit" delta value

- A new item – what to do ?

- If it has a counter, increment counter
- Otherwise, initiallize with a count of 1 + delta
- Whenever delta increases :
  - Delete tuples with a count smaller than delta

Manku & Motwani – Lossy-Counting

- Deleting tuples reduces the required space !
- Monitored items can have their frequencies overestimated by no more than $n / k = \varepsilon \times n$
- BUT never underestimated !!

The Metwally et al. algorithm

Algorithm 3: SpaceSaving(k)

```
n ← 0;
T ← ∅;
foreach i do
    n ← n + 1;
    if i ∈ T then c_i ← c_i + 1;
    else if |T| < k then
        T ← T ∪ {i};
        c_i ← 1;
    else
        j ← arg min_{j∈T} c_j;
        c_i ← c_j + 1;
        T ← T ∪ {i} \ {j};
```

[Cormode and Hadjieleftheriou]

Metwally et al. – Space-Saving

- Keep k=1/ε item names and counts
  - Initially zero
- Count first k items exactly !
- A new item – what to do ?

- If it has a counter, increment counter
- Otherwise, replace item with least count
- And increment count

Metwally et al. – Space-Saving

- Counters sum to n !
- Average count value is $n / k = \varepsilon \times n$
  - Smallest count min cannot be larger than $\varepsilon \times n$
- True count of an uncounted item is between 0 and $\varepsilon \times n$
- All items whose true count is $> \varepsilon \times n$ are stored !

Tasks

- Implement the Lossy-Counting and the Space-Saving algorithms
- Test them, using the same input sequences as for the M & G algorithm
- Compare the behavior of the three algorithms

Implementation issues

- Similar to Misra & Gries
- Finding the min item is a standard problem
  - Use a min-heap !
  - Binary, binomial, Fibonacci, … ?
  - O(log k)

Question

- What can you say about the estimated counts for items which are stored by the algorithms early in the stream and are not removed ?

Question

- Have we been discussing deterministic algorithms or randomized/probabilistic algorithms ?

Experimental comparison

- Cormode & Hadjieleftheriou
  - VLDB 2008 - https://dl.acm.org/citation.cfm?id=1454225
  - CACM 2009 - https://dl.acm.org/citation.cfm?id=1562789
- SPACESAVING has benefits over others !
- Very fast: 20M – 30M updates *per second*
- Implementation choices: speed vs space
  - E.g., a heap or lists of items grouped by frequencies

2017 – Recent progress

A High-Performance Algorithm for Identifying Frequent Items in Data Streams

Daniel Anderson
Georgetown University

Pryce Bevan
Georgetown University

Kevin Lang
Oath Research

Edo Liberty*
Amazon

Lee Rhodes
Oath

Justin Thaler
Georgetown University

ABSTRACT

Estimating frequencies of items over data streams is a common building block in streaming data measurement and analysis. Misra and Gries introduced their seminal algorithm for the problem in 1982, and the problem has since been revisited many times due its practicality and applicability. We describe a highly optimized version of Misra and Gries' algorithm that is suitable for deployment in industrial settings. Our code is made public via an open source library called Data Sketches that is already used by several companies and production systems.

been studied intensely [6, 7, 9, 13, 14, 17, 21, 31–35]. These algorithms process a massive dataset in a single pass, and compute very small *summaries* of the dataset, from which it is possible to derive accurate—though approximate—answers to frequent items queries and point queries.

It may seem as though streaming frequency approximation is well-understood, with little room for further insight or improvement. However, when we set about implementing an algorithm suitable for industrial use on web-scale data, we found that existing algorithms have two significant shortcomings. First, they are not

*Research performed while at Yahoo Research.

https://dl.acm.org/citation.cfm?doid=3131365.3131407

https://datasketches.github.io/

References

- R. Boyer & J. Moore, MJRTY – A fast majority vote algorithm, in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Springer, 1991

- J. Misra & D. Gries, Finding repeated elements, *Science of Computer Programming*, Vol. 2, 1982

- G. Cormode & M. Hadjieleftheriou, Finding the frequent items in streams of data, *Commun. ACM*, Vol. 52, N. 10, 2009