Tribhuvan University
Institute of Engineering
Purwanchal Campus
पूर्वाञ्चल क्याम्पस

7. Streaming Algorithms

Pukar Karki
Assistant Professor

Streaming Algorithms

- In several emerging applications, data takes the form of continuous data streams, as opposed to finite stored datasets.
- Examples include stock tickers, network traffic measurements, web-server logs, click streams, data feeds from sensor networks, and telecom call records.
- Stream processing differs from computation over traditional stored datasets in two important aspects:

  (a) the sheer volume of a stream over its lifetime could be huge, and

  (b) queries require timely answers; response times should be small.

Streaming Algorithms

- As a consequence, it is not possible to store the stream in its entirety on secondary storage and scan it when a query arrives.
- This motivates the design for summary data structures with small memory footprints that can support both one-time and continuous queries.

Why Streaming Algorithms?

- Networks

  - Up to 1 Billion packets per hour per router. Each ISP has hundreds of routers.

  - Spot faults, drops, failures

- Genomics

  - Whole genome sequences for many species now available, each megabytes to gigabytes in size

  - Analyse genomes, detect functional regions, compare across species

- Telecommunications

  - There are 3 Billion Telephone Calls in US each day, 30 Billion emails daily, 1 Billion SMS, IMs

  - Generate call quality stats, number/frequency of dropped calls

- Infeasible to store all this data in random access memory for processing.
- Solution – process the data as a stream – streaming algorithms

Models

Data Stream Model

- In the data stream model, some or all of the input is represented as a finite sequence of integers (from some finite domain) which is generally not available for random access, but instead arrives one at a time in a "stream".
- If the stream has length n and the domain has size m, algorithms are generally constrained to use space that is logarithmic in m and n.
- They can generally make only some small constant number of passes over the stream, sometimes just one.

Models

Turnstile and cash register models

- Much of the streaming literature is concerned with computing statistics on frequency distributions that are too large to be stored.
- For this class of problems, there is a vector $\mathbf{a} = (a_1 , \ldots , a_n)$ (initialized to the zero vector 0)that has updates presented to it in a stream. The goal of these algorithms is to compute functions of **a** using considerably less space than it would take to represent a precisely. There are two common models for updating such streams, called the "cash register" and "turnstile" models.

Models

Turnstile and cash register models(Cont..)

- In the **cash register model**, each update is of the form $\langle$ **i , c** $\rangle$ so that $\mathbf{a_i}$ is incremented by some positive integer **c**. A notable special case is when **c = 1** (only unit insertions are permitted).
- In the **turnstile model**, each update is of the form $\langle$ **i , c** $\rangle$, so that $\mathbf{a_i}$ is incremented by some (possibly negative) integer **c.** In the "strict turnstile" model, no $\mathbf{a_i}$ at any time may be less than zero.

Models

Sliding window model

- Several papers also consider the "sliding window" model.
- In this model, the function of interest is computing over a fixed-size window in the stream.
- As the stream progresses, items from the end of the window are removed from consideration while new items from the stream take their place.

Evaluation

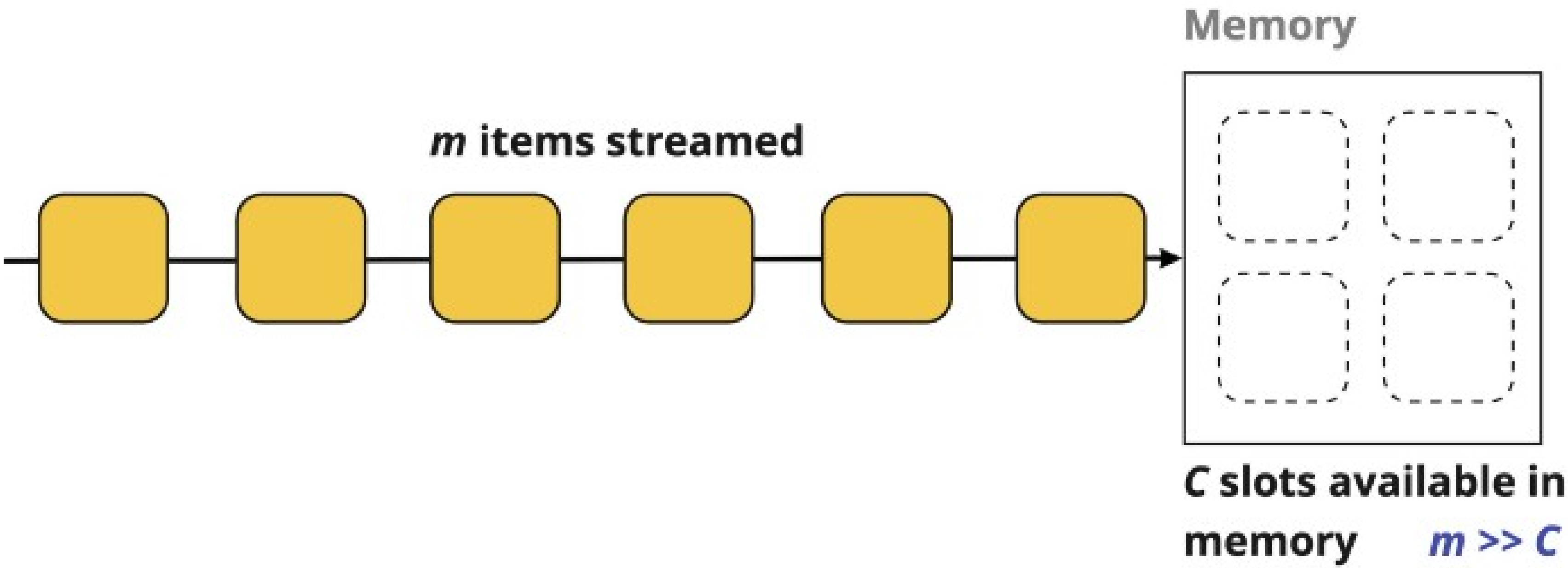
The performance of an algorithm that operates on data streams is measured by three basic factors:

- The number of passes the algorithm must make over the stream.
- The available memory.
- The running time of the algorithm.

Misra-Gries Summaries

- The frequent items problem is to process a stream of items and find all items occurring more than a given fraction of the time.
- Misra–Gries summaries are used to solve the frequent elements problem in the data stream model.
- That is, given a long stream of input that can only be examined once (and in some arbitrary order), the Misra-Gries algorithm can be used to compute which (if any) value makes up a majority of the stream, or more generally, the set of items that constitute some fixed fraction of the stream.

Misra-Gries Summaries



Misra-Gries Summaries

```
algorithm misra-gries:
    input:
        A positive integer k
        A finite sequence s taking values in the range 1,2,...,m
    output: An associative array A with frequency estimates for each item in s

    A := new (empty) associative array
    while s is not empty:
        take a value i from s
        if i is in keys(A):
            A[i] := A[i] + 1
        else if |keys(A)| < k - 1:
            A[i] := 1
        else:
            for each K in keys(A):
                A[K] := A[K] - 1
                if A[K] = 0:
                    remove K from keys(A)
    return A
```

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Initially,

Stream Value	Key	Value
-	-	

Note that 4 is appearing 5 times in the data stream which is more than n/k=4 times and thus should appear as the output of the algorithm.

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 1

Stream Value	Key	Value
1	1	1

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 4

Stream Value	Key	Value
4	-	0

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 5

Stream Value	Key	Value
5	5	1

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 4

Stream Value	Key	Value
4	-	0

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 4

Stream Value	Key	Value
4	4	1

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 5

Stream Value	Key	Value
5	-	0

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 4

Stream Value	Key	Value
4	4	1

Example

- Let k=2 and the data stream be 1,4,5,4,4,5,4,4 (n=8,m=5).
- Since k=2 and |keys(A)|=k−1=1 the algorithm can only have one key with its corresponding value.
- The algorithm will then execute as follows
- Read 4

Stream Value	Key	Value
4	4	2

Output: 4

Implementation

MisraGries.py - D:\IOE\Design and Analysis of Algorithms(CS924EE)\Practical\Chapter 7\MisraGries.py (3.10.4)

File Edit Format Run Options Window Help

```python
def MisraGries(S, k):
    A = dict()
    for value in S:
        if value in A.keys():
            A[value] = A[value]+1
        elif len(A.keys()) < k - 1:
            A[value] = 1
        else:
            for key in list(A):
                A[key] = A[key] - 1
                if A[key] == 0:
                    del A[key]

    return A

print(MisraGries([1,4,5,4,4,5,4,4], 2))
```

Output

```
{4: 2}
```

Analysis

- There is at most k-1 counters in D (that we can simplify to k). For each counter, we hold a key that can be from 1 to n, and a corresponding value that can be from 1 to m.
- Storing a key n require log(n) space (think of binary representations), and a counter m requires log(m) space. So one key-value pair represent log(n)+log(m) space.
- Since we have k-1 keys, we end up with a higher bound **$O(k*(\log_2 m+\log_2 n))$** for the algorithm space usage.

Count–Min Sketch

- Count-Min Sketch is a data structure for summarizing data streams.
- It allows fundamental queries in data stream summarization to be approximately answered very quickly
- In addition, it can be applied to solve several important problems in data streams such as finding quantiles, frequent items, etc.

Count–Min Sketch

- In an ideal case for retrieving frequency of any streaming data we use Hash Table as we can Store the Hash Values in the Hash table and retrieve them easily at O(1).
- But by doing so we are storing all the data in the hash tables which will fall in to super linear Memory usage for very large (infinite) streaming of data.

Count–Min Sketch

- In an ideal case for retrieving frequency of any streaming data we use Hash Table as we can Store the Hash Values in the Hash table and retrieve them easily at O(1).
- But by doing so we are storing all the data in the hash tables which will fall in to super linear Memory usage for very large (infinite) streaming of data.

Count–Min Sketch

- To tackle this memory in efficiency model , we can use count min sketch to calculate the frequency in sub-linear space , because in this case we will not be storing the complete values of data stream , instead we will use a matrix to compute the frequency, where number of rows would be number of Hash functions we are using and columns would be number of outcome of the Hash Functions.

Count–Min Sketch

- Lets say we have a stream of data Stream = {A,A,B,A,B,D,A…….}
- Lets define 4 hash functions H1,H2,H3,H4 and lets assume the below table for their outputs as shown in figure.

H1	1	3	6	2	5
H2	3	5	2	1	4
H3	1	3	5	4	2
H4	2	1	3	4	5

- Now lets create a matrix to keep a track of count of input streams:

Matrix -1	1	2	3	4	5	6
H1	0	0	0	0	0	0
H2	0	0	0	0	0	0
H3	0	0	0	0	0	0
H4	0	0	0	0	0	0

Count–Min Sketch

- Lets say we have a stream of data Stream = {**A**,A,B,A,B,D,A…….}
- Now for each data from stream now lets calculate the Hash outputs and increment the corresponding counter in the table….

H1(A) = 1, H2(A) = 3, H3(A) = 1, H4(A)=2

- Now lets create a matrix to keep a track of count of input streams:

Matrix -2	1	2	3	4	5	6
H1	1	0	0	0	0	0
H2	0	0	1	0	0	0
H3	1	0	0	0	0	0
H4	0	1	0	0	0	0

Count–Min Sketch

- Lets say we have a stream of data Stream = {A,**A**,B,A,B,D,A…….}
- Now for each data from stream now lets calculate the Hash outputs and increment the corresponding counter in the table….

H1(A) = 1, H2(A) = 3, H3(A) = 1, H4(A)=2

- Now lets update the matrix to keep a track of count of input streams:

Matrix -3	1	2	3	4	5	6
H1	2	0	0	0	0	0
H2	0	0	2	0	0	0
H3	2	0	0	0	0	0
H4	0	2	0	0	0	0

Count–Min Sketch

- Lets say we have a stream of data Stream = {A,A,**B**,A,B,D,A…….}
- Next in the stream we have B. So the Hash output of B is

  H1(B)= 3 ,H2(B)= 5 , H3(B) = 3 , H4(B) =1

- Now lets update the matrix to keep a track of count of input streams:

Matrix -4	1	2	3	4	5	6
H1	2	0	1	0	0	0
H2	0	0	2	0	1	0
H3	2	0	1	0	0	0
H4	1	2	0	0	0	0

Count–Min Sketch

- Lets say we have a stream of data Stream = {A,A,B,**A**,B,D,A…….}
- Next in the stream we have A. So the Hash output of A is

  H1(A) = 1, H2(A) = 3, H3(A) = 1, H4(A)=2

- Now lets update the matrix to keep a track of count of input streams:

Matrix -5	1	2	3	4	5	6
H1	3	0	1	0	0	0
H2	0	0	3	0	1	0
H3	3	0	1	0	0	0
H4	1	3	0	0	0	0

Count–Min Sketch

- Lets say we have a stream of data Stream = {A,A,B,A,**B**,D,A…….}
- Next in the stream we have B. So the Hash output of B is

  H1(B)= 3 ,H2(B)= 5 , H3(B) = 3 , H4(B) =1

- Now lets update the matrix to keep a track of count of input streams:

Matrix -6	1	2	3	4	5	6
H1	3	0	2	0	0	0
H2	0	0	3	0	2	0
H3	3	0	2	0	0	0
H4	2	3	0	0	0	0

Count–Min Sketch

- Lets say we have a stream of data Stream = {A,A,B,A,B,**D**,A…….}
- Next in the stream we have D. So the Hash output of D is

  H1(D)= 2 ,H2(D)= 1 , H3(D) = 4 , H4(D) =4
- Now lets update the matrix to keep a track of count of input streams:

Matrix -7	1	2	3	4	5	6
H1	3	1	2	0	0	0
H2	1	0	3	0	2	0
H3	3	0	2	1	0	0
H4	2	3	0	1	0	0

Count–Min Sketch

- Lets say we have a stream of data Stream = {A,A,B,A,B,D,**A**…….}
- Next in the stream we have A. So the Hash output of A is

  H1(A) = 1, H2(A) = 3, H3(A) = 1, H4(A)=2

- Now lets update the matrix to keep a track of count of input streams:

Matrix -8	1	2	3	4	5	6
H1	4	1	2	0	0	0
H2	1	0	4	0	2	0
H3	4	0	2	1	0	0
H4	2	4	0	1	0	0

Count–Min Sketch

- Now lets calculate the frequency of A…
- Again pass A to all hash functions and result is H1(A) = 1, H2(A) = 3, H3(A) = 1, H4(A)=2
- Now take the array of these positions in matrix which comes to (4,4,4,4) .. so minimum of this comes to 4 so the frequency of A= 4.

Matrix -8	1	2	3	4	5	6
H1	4	1	2	0	0	0
H2	1	0	4	0	2	0
H3	4	0	2	1	0	0
H4	2	4	0	1	0	0

Count–Min Sketch

- Similarly lets calculate frequency of B, H1(B)= 3 ,H2(B)= 5 , H3(B) = 3 , H4(B) =1.
- So the frequency = min (2,2,2,2) = 2

Matrix -8	1	2	3	4	5	6
H1	4	1	2	0	0	0
H2	1	0	4	0	2	0
H3	4	0	2	1	0	0
H4	2	4	0	1	0	0

Count–Min Sketch

- In some cases due to hash collision we might get the frequency little more than what is expected to come, hence it guarantees to give the exact frequency or more.
- The accuracy will depend upon how unique the hash functions return the value and also, more the number of hash functions, more accurate will the frequency be.
- In this way Count-Min sketch allows to calculate frequency of large data streams in sub linear space using same O(1) constant time complexity.

Locality Sensitive Hashing

- Locality-sensitive hashing (LSH) is an algorithmic technique that hashes similar input items into the same "buckets" with high probability.
- Locality-Sensitive Hashing (LSH) is a method which is used for determining which items in a given set are similar.
- Rather than using the naive approach of comparing all pairs of items within a set, items are hashed into buckets, such that similar items will be more likely to hash into the same buckets.
- As a result, the number of comparisons needed will be reduced; only the items within any one bucket will be compared.

Locality Sensitive Hashing

- Locality-sensitive hashing is often used when there exist an extremely large amount of data items that must be compared.
- In these cases, it may also be that the data items themselves will be too large, and as such will have their dimensionality reduced by a feature extraction technique beforehand.

Locality Sensitive Hashing

- The main application of LSH is to provide a method for **efficient approximate nearest neighbor search** through probabilistic dimension reduction of high-dimensional data.
- This dimensional reduction is done through feature extraction realized through hashing, for which different schemes are used depending upon the data.

Locality Sensitive Hashing

- LSH is used in fields such as data mining, pattern recognition, computer vision, computational geometry, and data compression.
- It also has direct applications in spell checking, plagiarism detection, and chemical similarity.

Locality Sensitive Hashing

Objectives: How to find efficiently

1. Similar documents among a collection of documents
2. Similar web-pages among web-pages
3. Similar fingerprints among a database of fingerprints
4. Similar sets among a collection of sets
5. Similar images from a database of images

Similarity of Documents

Problem Definition

- **Input:** A collection of web-pages.
- **Output:** Report near duplicate web-pages.

- **K-shingles:** Any substring of k words that appears in the document.

- **Text Document** = “What is the likely date that the regular classes may resume in Dharan”
- **2−shingles:** What is, is the, the likely, . . . , in Dharan
- **3−shingles:** What is the, is the likely, . . . , resume in Dharan
- **In practice:** 9−shingles for English Text and 5−shingles for e-mails

Similarity Between Sets

Text Document D → Set S

1. Form all the k-shingles of D
2. S is the collection of all k-shingles of D

Jaccard Similarity

- For a pair of sets S and T , the Jaccard Similarity is defined as

$$\mathsf{SIM}(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

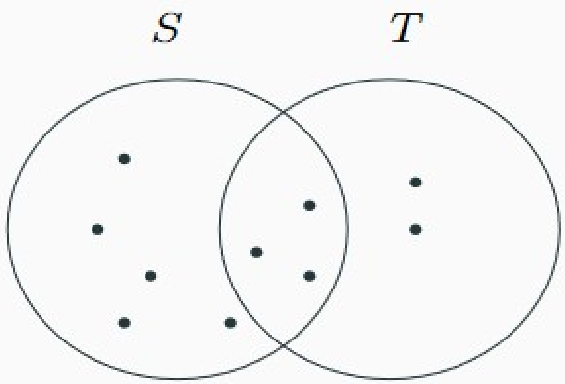


Figure : $|S| = 8, |T| = 5, |S \cup T| = 9, |S \cap T| = 3, \mathsf{SIM}(S,T) = \frac{|S \cap T|}{|S \cup T|} = \frac{3}{9} = \frac{1}{3}$

Problem: Find Similar Sets

New Problem

Given a constant $0 \le s \le 1$ and a collection of sets S, find the pairs of sets in S with Jaccard similarity $\ge s$?

U = {Cruise, Ski, Resorts, Safari, Stay@Home}

S_1 = {Cruise, Safari}

S_2 = {Resorts}

S_3 = {Ski, Safari, Stay@Home}

S_4 = {Cruise, Resorts, Safari}

Problem: Given S = {S_1, S_2, S_3, S_4} and s = 1/2 , report all pairs that are s-similar.

$\mathsf{SIM}(S_1, S_3) = \frac{1}{4}$ $\mathsf{SIM}(S_2, S_4) = \frac{1}{3}$

$\mathsf{SIM}(S_1, S_4) = \frac{2}{3}$ $\mathsf{SIM}(S_3, S_4) = \frac{1}{5}$

Characteristic Matrix Representation of Sets

U = {Cruise, Ski, Resorts, Safari, Stay@Home}

$S = \{S_1, S_2, S_3, S_4\}$, where each $S_i \subseteq U$

e.g. S_1 = {Cruise, Safari} and S_2 = {Resorts}

Characteristic matrix for S:

	S_1	S_2	S_3	S_4
Cruise	1	0	0	1
Ski	0	0	1	0
Resorts	0	1	0	1
Safari	1	0	1	1
Stay@Home	0	0	1	0

MinHash Signatures via Random Permutation

Permute Rows of characteristic matrix - π : 01234 → 40312

		S_1	S_2	S_3	S_4
0	Cruise	1	0	0	1
1	Ski	0	0	1	0
2	Resorts	0	1	0	1
3	Safari	1	0	1	1
4	Stay@Home	0	0	1	0

		S_1	S_2	S_3	S_4
0(1)	Ski	0	0	1	0
1(3)	Safari	1	0	1	1
2(4)	Stay@Home	0	0	1	0
3(2)	Resorts	0	1	0	1
4(0)	Cruise	1	0	0	1

Minhash Signatures for a set S_i w.r.t. π is the **row-number** of first non-zero element in the column corresponding to S_i

$h(S_1) = 1$

$h(S_2) = 3$

$h(S_3) = 0$

$h(S_4) = 1$

Key Lemma

Lemma

For any two sets S_i and S_j in a collection of sets S where the elements are drawn from the universe U , the probability that the minhash value $h(S_i)$ equals $h(S_j)$ is equal to the Jaccard similarity of S_i and S_j , i.e.

$$Pr[h(S_i) = h(S_j)] = \mathrm{SIM}(S_i, S_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$$

		S_1	S_2	S_3	S_4
0	Ski	0	0	1	0
1	Safari	1	0	1	1
2	Stay@Home	0	0	1	0
3	Resorts	0	1	0	1
4	Cruise	1	0	0	1

$$Pr[h(S_1) = h(S_4)] = \mathrm{SIM}(S_1, S_4) = \frac{|S_1 \cap S_4|}{|S_1 \cup S_4|} = \frac{2}{3}$$

Proof of Key Observation

- Consider the rows corresponding to the columns of S_i and S_j .
- Let **x** = Number of rows where both the columns have a 1.
- Let **y** = Number of rows where exactly one of the columns has a 1
- Observe that $|S_i \cap S_j| =$ **x** and,

  $|S_i \cup S_j| =$ **x** + **y**.

S_1	S_4		
0	0		
1	1	$\rightarrow$	x
0	0		
0	1	$\rightarrow$	y
1	1	$\rightarrow$	x

- Note that the rows where both the columns have 0's can't be the minHash signature of S_i or S_j .
- Probability that $h(S_i) = h(S_j)$ is same as that the row corresponding to x is the 'first one' as compared to the rows corresponding to y.

$$\text{Thus, } Pr[h(S_i) = h(S_j)] = \frac{x}{x+y} = \frac{|S_i \cap S_j|}{|S_i \cup S_j|} = \mathrm{SIM}(S_i, S_j)$$

MinHashSignature Matrix

MinHash Signature matrix for |S| = 11 sets with 12 hash functions

S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
2	2	1	0	0	1	3	2	5	0	3
1	3	2	0	2	2	1	4	2	1	2
3	0	3	0	4	3	2	0	0	4	2
0	4	3	1	5	3	3	2	3	5	4
2	1	1	0	4	1	2	1	4	2	5
4	2	1	0	5	2	3	2	3	5	4
2	4	3	0	5	3	3	4	4	5	3
0	2	4	1	3	4	3	2	2	2	4
0	2	1	0	5	1	1	1	1	5	1
0	5	1	0	2	1	3	2	1	5	4
1	3	1	0	5	2	3	3	6	3	2
0	5	2	1	5	1	2	2	6	5	4

LSH for MinHash

Partitioning of a signature matrix into b= 4 bands of r= 3 rows each.

Band	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
	2	2	1	0	0	1	3	2	5	0	3
I	1	3	2	0	2	2	1	4	2	1	2
	3	0	3	0	4	3	2	0	0	4	2
	0	4	3	1	5	3	3	2	3	5	4
II	2	1	1	0	4	1	2	1	4	2	5
	4	2	1	0	5	2	3	2	3	5	4
	2	4	3	0	5	3	3	4	4	5	3
III	0	2	4	1	3	4	3	2	2	2	4
	0	2	1	0	5	1	1	1	1	5	1
	0	5	1	0	2	1	3	2	1	5	4
IV	1	3	1	0	5	2	3	3	6	3	2
	0	5	2	1	5	1	2	2	6	5	4

Band 3: $\{S_3, S_6, S_{11}\}$ are hashed into the same bucket, and so are $\{S_8, S_9\}$

Probability of Finding Similar Sets

Lemma

Let s > 0 be the Jaccard similarity of two sets. The probability that the minHash signature matrix agrees in all the rows of at least one of the bands for these two sets is

$$f(s) = 1 - (1 - s^r)^b$$

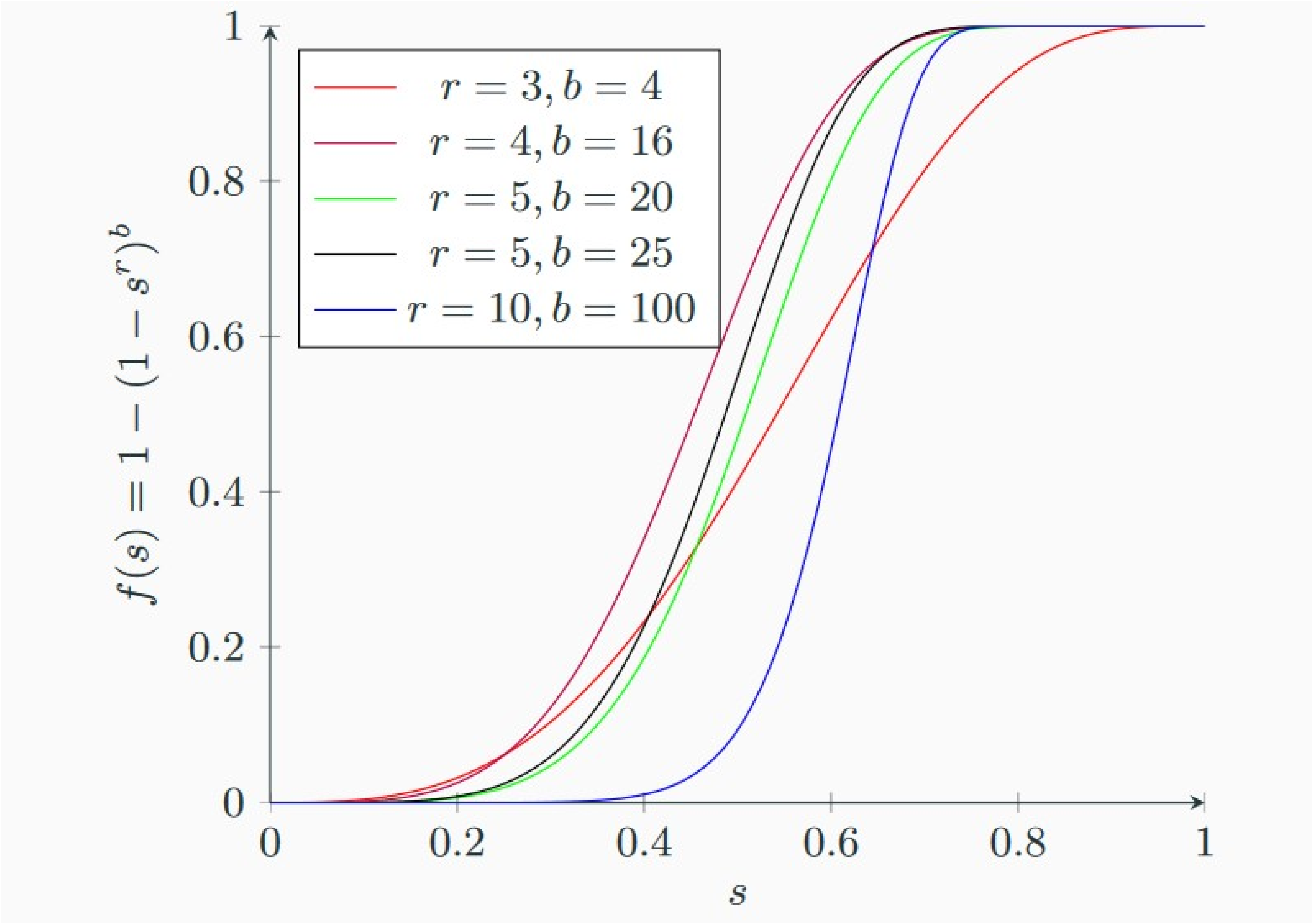
Band	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
	2	2	1	0	0	1	3	2	5	0	3
I	1	3	2	0	2	2	1	4	2	1	2
	3	0	3	0	4	3	2	0	0	4	2
	0	4	3	1	5	3	3	2	3	5	4
II	2	1	1	0	4	1	2	1	4	2	5
	4	2	1	0	5	2	3	2	3	5	4
	2	4	3	0	5	3	3	4	4	5	3
III	0	2	4	1	3	4	3	2	2	2	4
	0	2	1	0	5	1	1	1	1	5	1
	0	5	1	0	2	1	3	2	1	5	4
IV	1	3	1	0	5	2	3	3	6	3	2
	0	5	2	1	5	1	2	2	6	5	4

Understanding f(s)

$f(s) = 1 - (1 - s^r)^b$ for different values of s, b, and r:

(b, r) $f(s) = 1 - (1 - s^r)^b \searrow$	$(4, 3)$	$(16, 4)$	$(20, 5)$	$(25, 5)$	$(100, 10)$
$s = 0.2$	0.0316	0.0252	0.0063	0.0079	0.0000
$s = 0.4$	0.2324	0.3396	0.1860	0.2268	0.0104
$s = 0.5$	0.4138	0.6439	0.4700	0.5478	0.0930
$s = 0.6$	0.6221	0.8914	0.8019	0.8678	0.4547
$s = 0.8$	0.9432	0.9997	0.9996	0.9999	0.9999
$s = 1.0$	1.0	1.0	1.0	1.0	1.0
Threshold $t = (\frac{1}{b})^{(\frac{1}{r})}$	0.6299	0.5	0.5492	0.5253	0.6309

S-curve



Computational Summary

- **Input:** Collection of m text documents of size D
- **k-shingles:** Size = **k.D**
- Characteristic matrix of size **|U| × m**, where **U** is the universe of all possible k-shingles
- Signature matrix of size **n × m** using n-permutations
- **floor(n/r)** bands each consisting of r rows
- Hash maps from bands to buckets
- **Output:** All pairs of documents that are in the same bucket corresponding to a band
- Check whether the pairs correspond to similar documents!
- With the right choice of threshold **P_r(the pair is similar)** $\rightarrow$ 1

Lossy Count Algorithm

- The **lossy count** algorithm is an algorithm to identify elements in a data stream whose frequency count exceed a user-given threshold.
- The frequency computed by this algorithm is not always accurate, but has an error threshold that can be specified by the user.
- The run time space required by the algorithm is inversely proportional to the specified error threshold, hence larger the error, the smaller the footprint.

Lossy Count Algorithm - Motivations

- Here are four problems drawn from databases, data mining, and computer networks, where frequency counts exceeding a user-specified threshold are computed.

1) **An iceberg query** performs an aggregate function over an attribute (or a set of attributes) of a relation and eliminates those aggregate values that are below some user-specified threshold.
2) **Association rules** over a dataset consisting of sets of items, require computation of frequent itemsets, where an itemset is frequent if it occurs in at least a user-specified fraction of the dataset.
3) **Iceberg datacubes** compute only those Group By's of a CUBE operator whose aggregate frequency exceeds a user-specified threshold.
4) **Traffic measurement and accounting of IP packets** requires identification of flows that exceed a certain fraction of total traffic.

Lossy Count Algorithm - Motivations

- Existing algorithms for iceberg queries, association rules, and iceberg cubes have been optimized for finite stored data.
- They compute exact results, attempting to minimize the number of passes they make over the entire dataset.
- The best algorithms take two passes.

Lossy Count Algorithm - Motivations

- When adapted to streams, where only one pass is allowed, and results are always expected to be available with short response times, these algorithms fail to provide any a priori guarantees on the quality of their output.
- Lossy Count Algorithm computes frequency counts in a single pass with a priori error guarantees.
- This algorithms work for variable sized transactions and can also compute frequent sets of items in a single pass.

Algorithm

- This algorithm accepts two user-specified parameters: a support threshold $\boldsymbol{s} \in \mathbf{(0,1)}$, and an error parameter $\boldsymbol{\varepsilon} \in \mathbf{(0,1)}$ such that $\boldsymbol{\varepsilon \ll s}$.
- Let **N** denote the current length of the stream, i.e., the number of tuples seen so far.
- At any point of time, this algorithm can produce a list of item(set)s along with their estimated frequencies. The answers produced by this algorithm will have the following guarantees:

1. All item(set)s whose true frequency exceeds $\boldsymbol{s.N}$ are output. There are no false negatives.
2. No item(set) whose true frequency is less than $\boldsymbol{(s - \varepsilon).N}$ is output.
3. Estimated frequencies are less than the true frequencies by at most $\boldsymbol{\varepsilon.N}$

This algorithm consumes at most $\boldsymbol{1/\varepsilon * \log(\varepsilon N)}$ space

Algorithm(Cont..)

- Imagine a user who is interested in identifying all items whose frequency is at least 0.1% of the entire stream seen so far.
- Then s = 0.1%
- The user is free to set **ε** what-ever she feels is a comfortable margin of error.
- Suppose, **ε** = 0.01%
- As per Property 1, all elements with frequency exceeding **s = 0.1%** will be output; there will be no false negatives.
- As per Property 2, no element with frequency below **(*s* – ε) = 0.09%** will be output.
- As per property 3, all individual frequencies are less than their true frequencies by at most **ε = 0.01%.**

Example

With $s = 10\%, \epsilon = 1\%, N = 1000$

Example

$$\text{With } s = 10\%, \epsilon = 1\%, N = 1000$$

1. All elements exceeding frequency $sN = 100$ will be output.
2. No elements with frequencies below $(s - \epsilon)N = 90$ are output. False positives between 90 and 100 might or might not be output.
3. All estimated frequencies diverge from their true frequencies by at most $\epsilon N = 10$ instances.

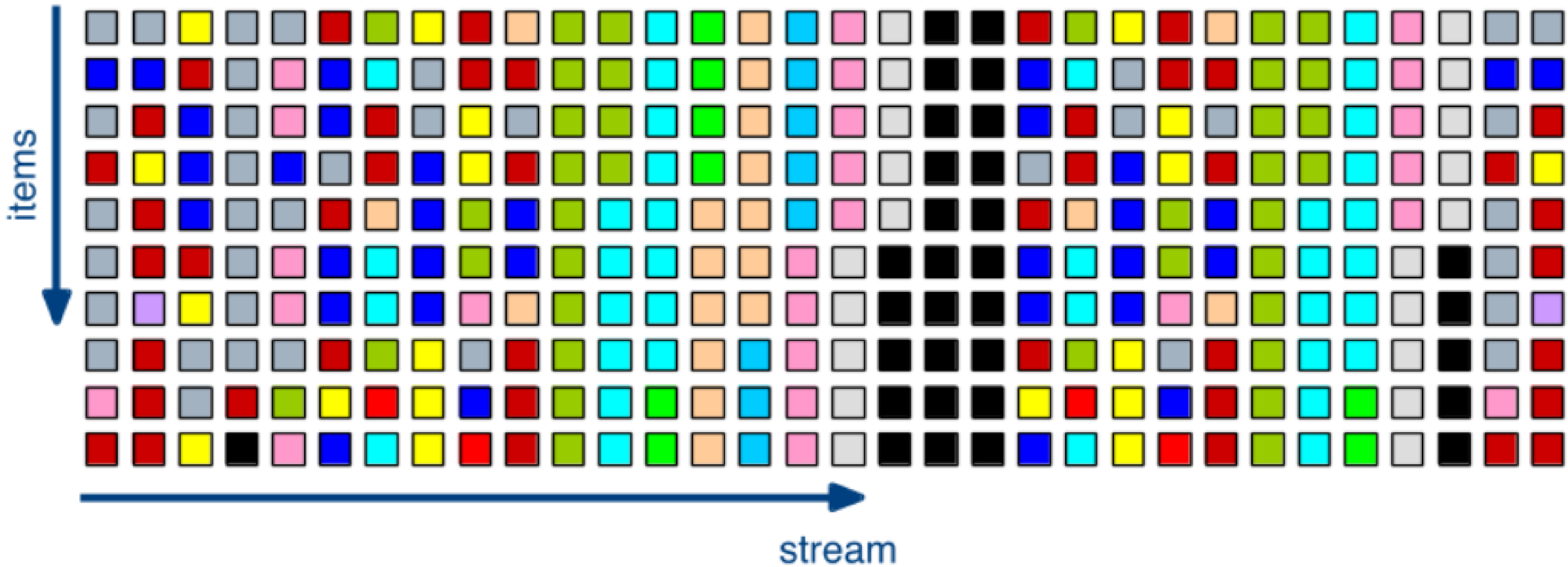
Rule of thumb: $\epsilon = 0.1s$

Expected Errors

1. high frequency false positives
2. small errors in frequency estimations

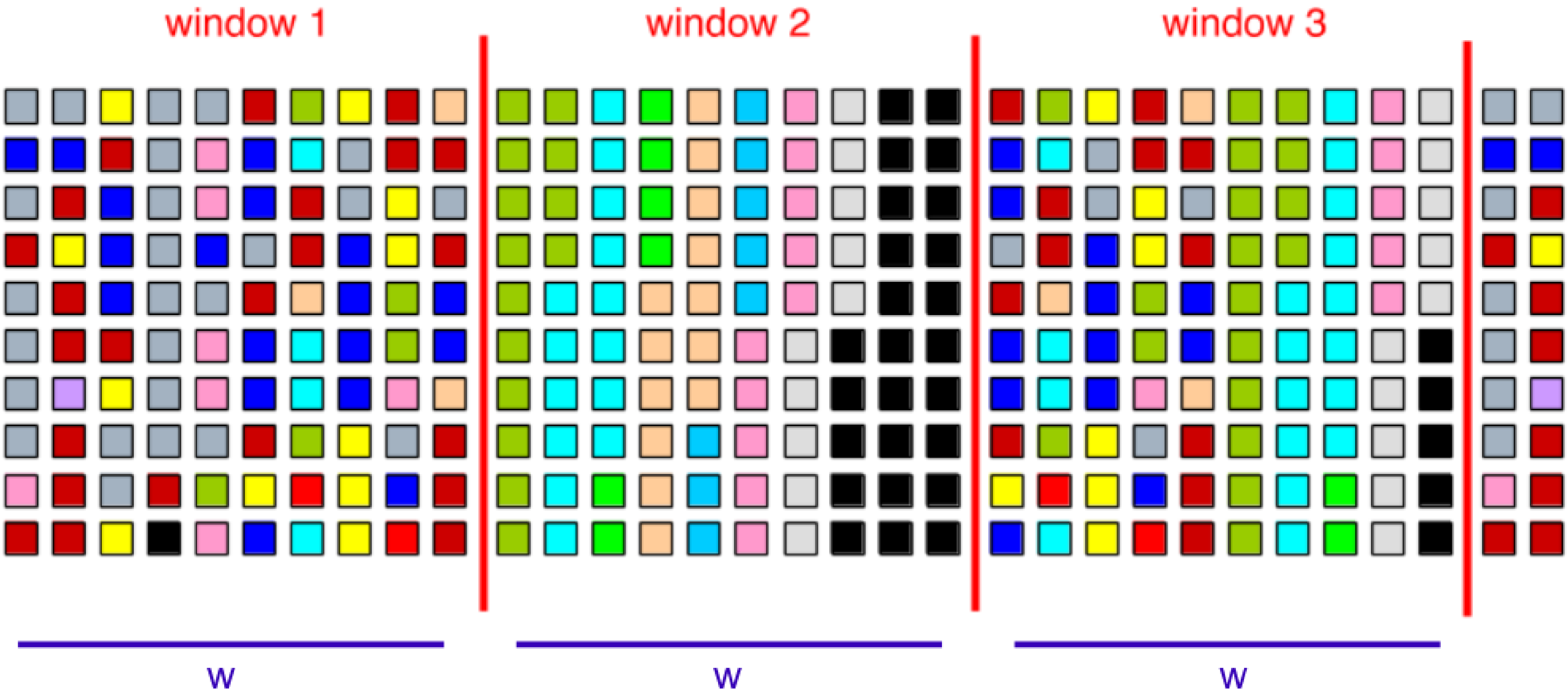
Acceptable for high numbers of N

Lossy Counting in Action



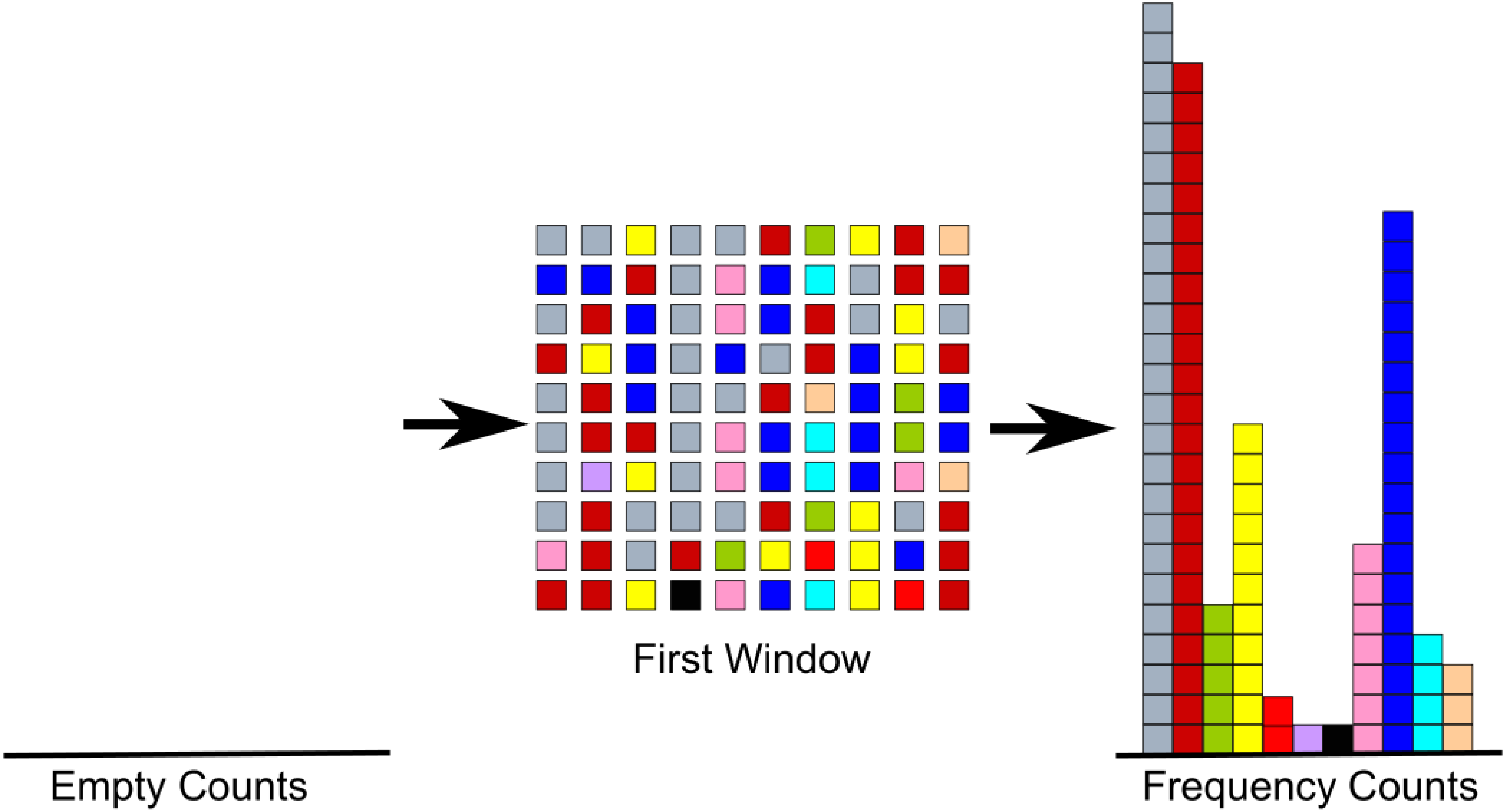
Incoming Stream of Colours

Divide into Windows/Buckets



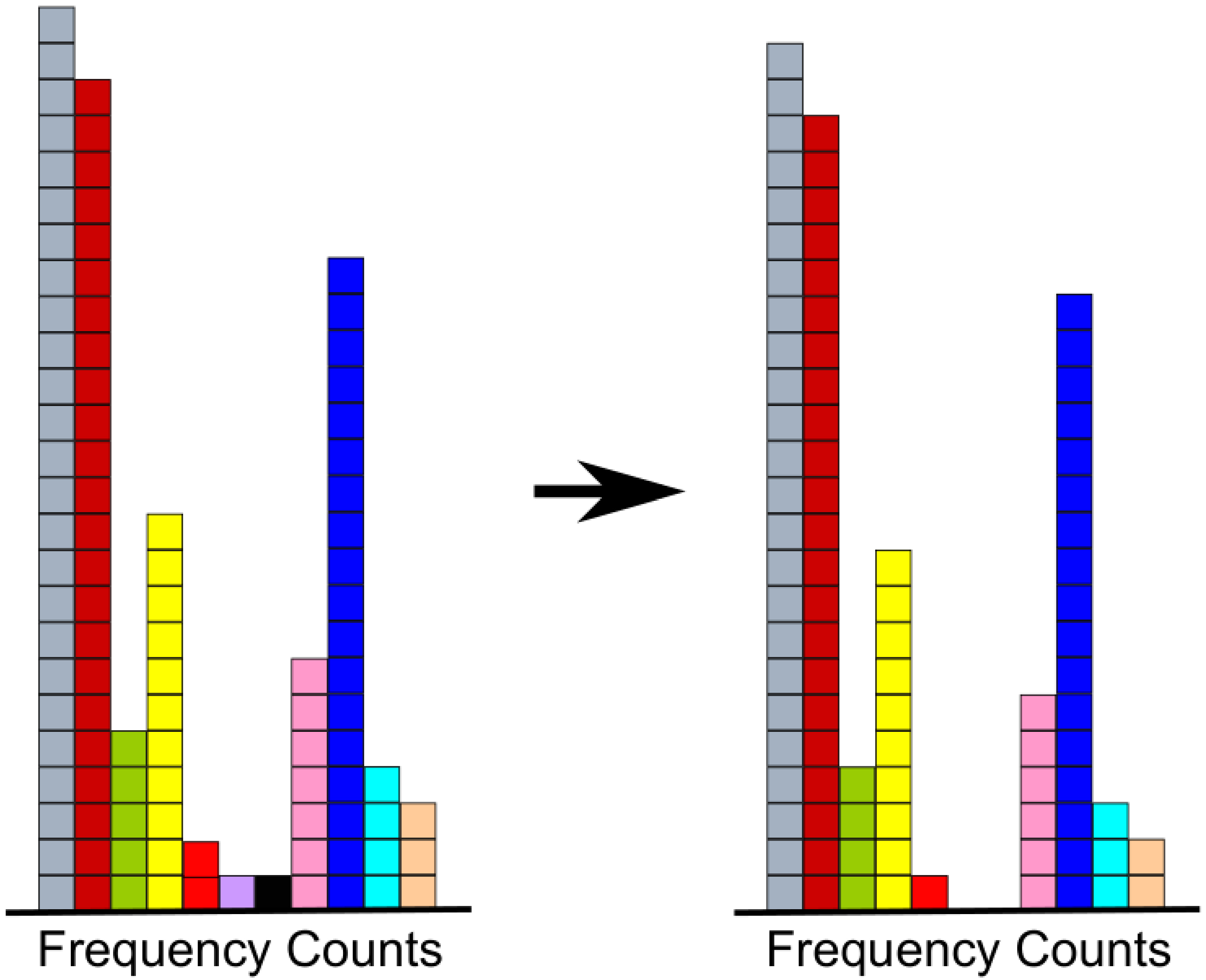
$$\text{Window Size } w = \left\lceil \frac{1}{\epsilon} \right\rceil = \left\lceil \frac{1}{0.01} \right\rceil = 100$$

First Window Comes In



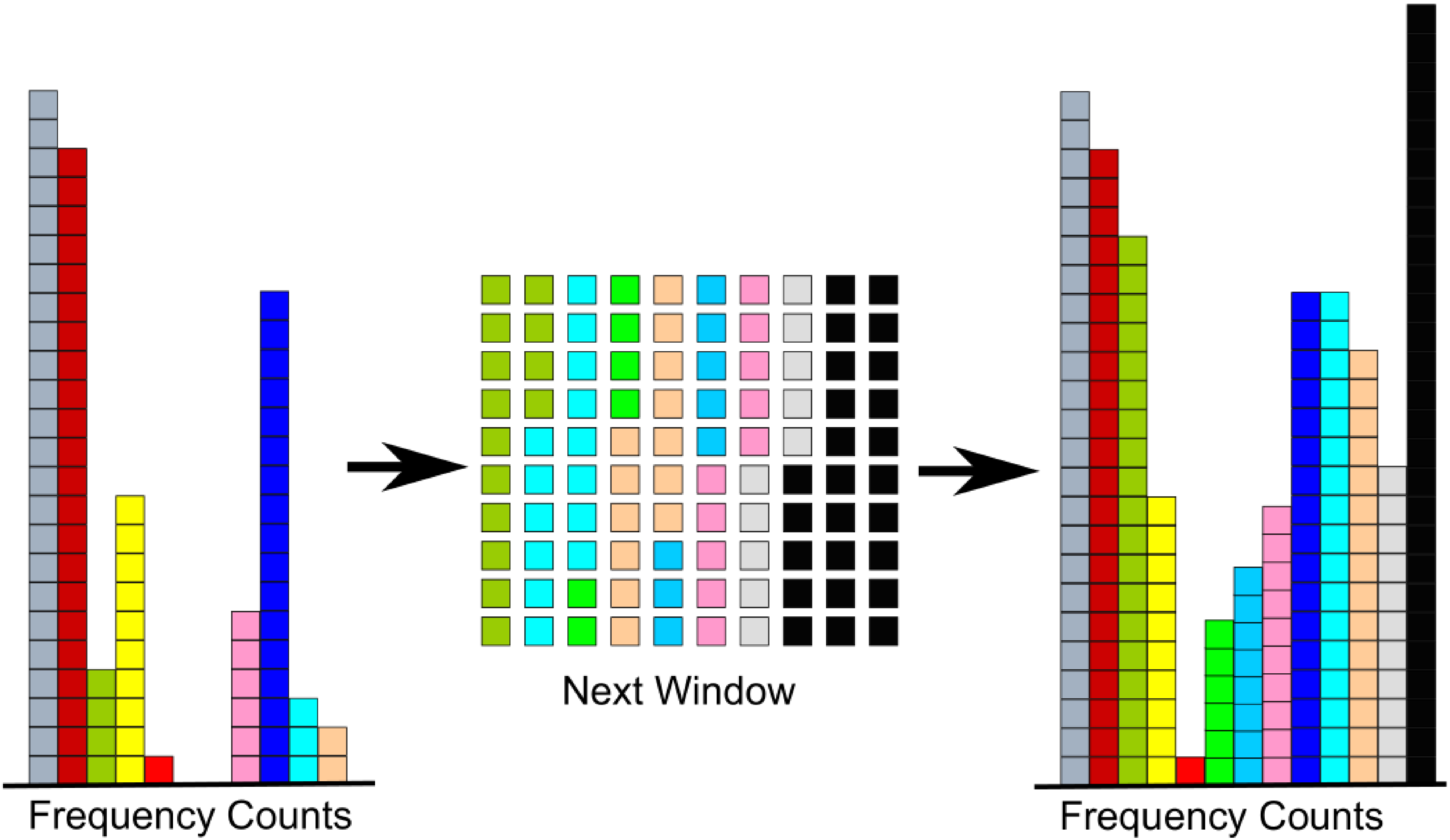
Go through elements. If counter exists, increase by one, if not create one and initialise it to one.

Adjust Counts at Window Boundaries



Reduce all counts by one. If counter is zero for a specific element, drop it.

Next Window Comes In



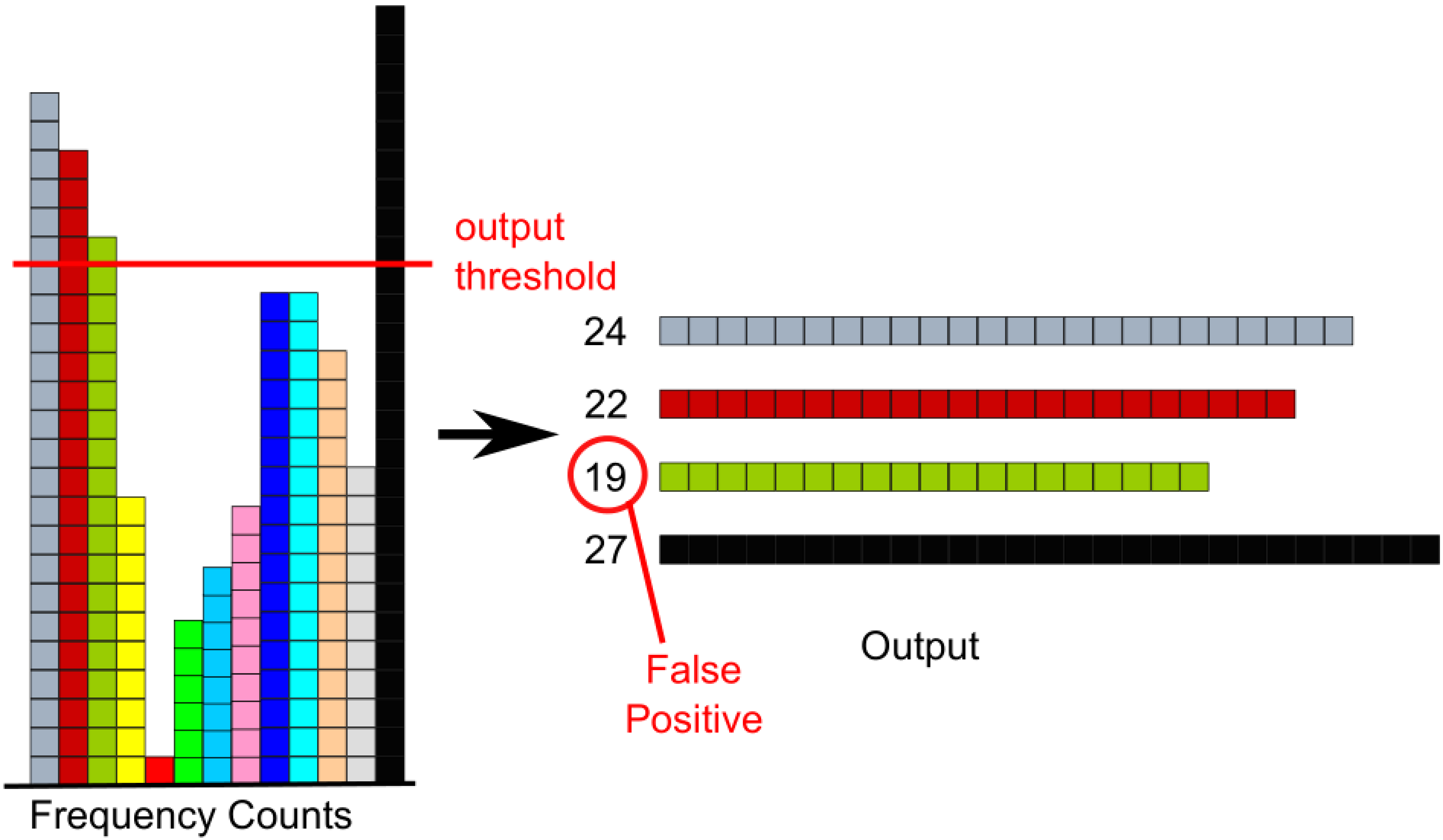
Count elements and adjust counts afterwards.

Lossy Counting Summary

- Split Stream into Windows
- For each window: Count elements, if no counter exists, create one.
- At window boundaries: Reduce all frequencies by one. If frequency goes to zero, drop counter.
- Process next window.

Output

With $s = 10\%, \epsilon = 1\%, N = 200$



To reduce false positives to acceptable amount, only output counters with frequency **$f \geq (s - \epsilon)N = 18$.**

Other Counting Algorithms Based on Stream Windows

Sticky Sampling

Lossy Counting vs. Sticky Sampling

Feature	Lossy Counting	Sticky Sampling
Results	deterministic	probabilistic
Memory	grows with N	static (independent of N)
Theory	performs worse	performs better
Practice	performs better	performs worse

performance in terms of memory and accuracy