CS 591 K1:
Data Stream Processing and Analytics

Spring 2020

4/23: Cardinality and frequency estimation

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Counting distinct elements

How can we count the number of **distinct elements** seen so far in a stream?

Example use-case: **Distinct users visiting one or multiple webpages**

How can we count the number of **distinct elements** seen so far in a stream?

Example use-case: **Distinct users visiting one or multiple webpages**

Naive solution: maintain a hash table

How can we count the number of **distinct elements** seen so far in a stream?

Example use-case: **Distinct users visiting one or multiple webpages**

Naive solution: maintain a hash table

Convert the stream into a multi-set of *uniformly distributed* random numbers using a *hash function*.

How can we count the number of **distinct elements** seen so far in a stream?

Example use-case: **Distinct users visiting one or multiple webpages**

Naive solution: maintain a hash table

Convert the stream into a multi-set of *uniformly distributed* random numbers using a *hash function*.

The more different elements we encounter in the stream, the more different hash values we shall see.

Let **h** be a hash function that maps each stream element into $M = log_2 N$ bits, where N is the domain of input elements:

$$h(x) = \sum_{k=0}^{M-1} i_k 2^k = (i_0 i_1 \ldots i_{M-1})_2, i_k \in \{0,1\}$$

For each element x, let $rank(x)$ be the number of *0s* in the end of $h(x)$:

- e.g.
  - x_1 = 318, $h(x_1)$ = 12 or 011**00** => $rank(x_1)$ = 2
  - x_2 = 9013, $h(x_2)$ = 24 or 11**000** => $rank(x_2)$ = 3

Let $\mathbf{n}$ be the number of distinct elements in the input stream so far and let R be the maximum value of *rank(.)* seen so far.

Let $\mathbf{n}$ be the number of distinct elements in the input stream so far and let R be the maximum value of $rank(.)$ seen so far.

Claim: The maximum observed rank is a good estimate of $\log_2 \mathbf{n}$.

In other words, the estimated number of distinct elements is equal to:

$$\hat{n} = 2^R$$

The hash function h hashes x to any of N values with probability $1/N$.

Out of all x we hash:

- around 50% will have a binary representation that ends in at least one 0:
  - ********0 (the probability of a 0 is 1/2)
- around 25% will end in at least two 0s:
  - *******00 (1/2 * 1/2)
- and so on…

The hash function h hashes x to any of N values with probability $1/N$.

Out of all x we hash:

- around 50% will have a binary representation that ends in at least one 0:
  - ********0 (the probability of a 0 is 1/2)
- around 25% will end in at least two 0s:
  - *******00 (1/2 * 1/2)
- and so on…

*If **one 0** is the maximum we've seen, that indicates **2 distinct elements**, whereas if **two 0s** is the maximum we've seen, that indicates **4 distinct elements**, …*

The hash function h hashes x to any of N values with probability $1/N$.

Out of all x we hash:

- around 50% will have a binary representation that ends in at least one 0:
  - ********0 (the probability of a 0 is 1/2)
- around 25% will end in at least two 0s:
  - *******00 (1/2 * 1/2)
- and so on…

*If **one 0** is the maximum we've seen, that indicates **2 distinct elements**, whereas if **two 0s** is the maximum we've seen, that indicates **4 distinct elements**, …*

It takes 2^r hash calls before we encounter a result with r 0s.

Is this a good estimate?

Is this a good estimate?

The probability that a given $h(x)$ ends in at least r 0s is:

$$\frac{1}{2} * \frac{1}{2} * \frac{1}{2} \ldots \frac{1}{2} = 2^{-r}$$

Is this a good estimate?

The probability that a given $h(x)$ ends in at least r 0s is:

$$\frac{1}{2} * \frac{1}{2} * \frac{1}{2} \dots \frac{1}{2} = 2^{-r}$$

The probability of *not* seeing a tail with at least r 0s among k elements is:

$$(1 - 2^{-r})^k$$

Is this a good estimate?

The probability that a given $h(x)$ ends in at least r 0s is:

$$\frac{1}{2} * \frac{1}{2} * \frac{1}{2} \dots \frac{1}{2} = 2^{-r}$$

The probability of *not* seeing a tail with at least r 0s among k elements is:

$$(1 - 2^{-r})^k$$

The probability that $h(x)$ ends in less then r 0s

Is this a good estimate?

The probability that a given $h(x)$ ends in at least r 0s is:

$$\frac{1}{2} * \frac{1}{2} * \frac{1}{2} \dots \frac{1}{2} = 2^{-r}$$

The probability of *not* seeing a tail with at least r 0s among k elements is:

$$(1 - 2^{-r})^k$$

for all k elements

The probability that $h(x)$ ends in less then r 0s

The probability of *not* seeing a tail with at least r 0s among k elements is $(1 - 2^{-r})^k$

The probability of *not* seeing a tail
with at least r 0s among k elements is $(1-2^{-r})^k$

We know that $(1-\epsilon)^{1/\epsilon} = 1/e$

The probability of *not* seeing a tail with at least r 0s among k elements is $(1-2^{-r})^k$

We know that $(1-\epsilon)^{1/\epsilon} = 1/e$

For $\epsilon = 2^{-r} \rightarrow (1-2^{-r})^k = e^{-k2^{-r}}$

The probability of *not* seeing a tail with at least r 0s among k elements is $(1-2^{-r})^k$

We know that $(1-\epsilon)^{1/\epsilon} = 1/e$

For $\epsilon = 2^{-r} \rightarrow (1-2^{-r})^k = e^{-k2^{-r}}$

- If $k \gg 2^r$: $\dfrac{k}{2^r} \rightarrow 0 \; and \; e^{-k2^{-r}} \rightarrow 1$

The probability of *not* seeing a tail
with at least r 0s among k elements is $(1 - 2^{-r})^k$

We know that $(1 - \epsilon)^{1/\epsilon} = 1/e$

For $\epsilon = 2^{-r} \rightarrow (1 - 2^{-r})^k = e^{-k2^{-r}}$

- If $k \gg 2^r$: $\dfrac{k}{2^r} \rightarrow 0 \; and \; e^{-k2^{-r}} \rightarrow 1$
- If $k \ll 2^r$: $\dfrac{k}{2^r} \rightarrow \infty \; and \; e^{-k2^{-r}} \rightarrow 0$

The probability of *not* seeing a tail with at least r 0s among k elements is $(1-2^{-r})^k$

We know that $(1-\epsilon)^{1/\epsilon} = 1/e$

For $\epsilon = 2^{-r} \rightarrow (1-2^{-r})^k = e^{-k2^{-r}}$

- If $k \gg 2^r$: $\frac{k}{2^r} \rightarrow 0$ *and* $e^{-k2^{-r}} \rightarrow 1$
- If $k \ll 2^r$: $\frac{k}{2^r} \rightarrow \infty$ *and* $e^{-k2^{-r}} \rightarrow 0$

The estimate 2^R cannot be too high or too low.

Is it good enough?

Is it good enough?

If we increase the number of 0s at the end of a hash value by 1, 2^R doubles!

- R = 4, 2^R = 16 distinct elements
- R = 5, 2^R = 32 distinct elements
- R = 6, 2^R = 64 distinct elements

No estimate in between powers of 2!

Is it good enough?

If we increase the number of 0s at the end of a hash value by 1, 2^R doubles!

- R = 4, 2^R = 16 distinct elements
- R = 5, 2^R = 32 distinct elements
- R = 6, 2^R = 64 distinct elements

No estimate in between powers of 2!

To get a better estimate, we need to use **multiple hash functions** and combine their estimates:

- Using many hash functions for a high-rate stream is expensive
- Finding many random and independent hash functions is difficult

Stochastic averaging

Stochastic averaging

Use one hash function to simulate many by splitting the hash value into two parts

Stochastic averaging

Use one hash function to simulate many by splitting the hash value into two parts

We split the input stream into $m = 2^p$ sub-streams $S_0, S_1, \ldots, S_{m-1}$

For every element x, we compute $h(x)$ and use the p first bits of the M-bit hash value to select a sub-stream and the next M-p bits to compute the $rank(.)$:

Stochastic averaging

Use one hash function to simulate many by splitting the hash value into two parts

We split the input stream into $m = 2^p$ sub-streams $S_0, S_1, \ldots, S_{m-1}$

For every element x, we compute $h(x)$ and use the p first bits of the M-bit hash value to select a sub-stream and the next M-p bits to compute the $rank(.)$:

For $h(x) = (i_0 i_1 \ldots i_{M-1})_2, i_k \in \{0,1\}$ we select one of m counters

COUNT[j], where $j = (i_0 i_1 \ldots i_{p-1})_2$

Stochastic averaging: example

Let $M = 5$, $p = 2$ and a hash function h_5 that maps elements to a binary representation of length 5.

We split the stream into $m = 2^p = 4$ sub-streams.

Consider the input elements {5, 14, 5, 2, 8, 1, …}

Stochastic averaging: example

Let $M = 5$, $p = 2$ and a hash function h_5 that maps elements to a binary representation of length 5.

We split the stream into $m = 2^p = 4$ sub-streams.

Consider the input elements {5, 14, 5, 2, 8, 1, …}

Substream	Address	Counter
S_0	00	
S_1	01	
S_2	10	
S_3	11	

Stochastic averaging: example

Let $M = 5$, $p = 2$ and a hash function h_5 that maps elements to a binary representation of length 5.

We split the stream into $m = 2^p = 4$ sub-streams.

Consider the input elements {5, 14, 5, 2, 8, 1, …}

- $x_1=5$, $h_5(5) = 00101$
- $x_2=14$, $h_5(14) = 10110$
- $x_3=5$, $h_5(5) = 00101$
- $x_4=2$, $h_5(2) = 01000$
- $x_5=8$, $h_5(8) = 00100$
- $x_6=1$, $h_5(1) = 11010$

Substream	Address	Counter
S_0	00	
S_1	01	
S_2	10	
S_3	11	

Stochastic averaging: example

Let $M = 5$, $p = 2$ and a hash function h_5 that maps elements to a binary representation of length 5.

We split the stream into $m = 2^p = 4$ sub-streams.

Consider the input elements {5, 14, 5, 2, 8, 1, …}

- $x_1=5$, $h_5(5) = 00101$
- $x_2=14$, $h_5(14) = 10110$
- $x_3=5$, $h_5(5) = 00101$
- $x_4=2$, $h_5(2) = 01000$
- $x_5=8$, $h_5(8) = 00100$
- $x_6=1$, $h_5(1) = 11010$

Substream	Address	Counter
S_0	00	0
S_1	01	
S_2	10	
S_3	11	

Stochastic averaging: example

Let $M = 5$, $p = 2$ and a hash function h_5 that maps elements to a binary representation of length 5.

We split the stream into $m = 2^p = 4$ sub-streams.

Consider the input elements {5, 14, 5, 2, 8, 1, …}

- $x_1=5$, $h_5(5) = 00101$
- $x_2=14$, $h_5(14) = 10110$
- $x_3=5$, $h_5(5) = 00101$
- $x_4=2$, $h_5(2) = 01000$
- $x_5=8$, $h_5(8) = 00100$
- $x_6=1$, $h_5(1) = 11010$

Substream	Address	Counter
S_0	00	0
S_1	01	
S_2	10	1
S_3	11	

Stochastic averaging: example

Let $M = 5$, $p = 2$ and a hash function h_5 that maps elements to a binary representation of length 5.

We split the stream into $m = 2^p = 4$ sub-streams.

Consider the input elements {5, 14, 5, 2, 8, 1, …}

- $x_1=5$, $h_5(5) = 00101$
- $x_2=14$, $h_5(14) = 10110$
- $x_3=5$, $h_5(5) = 00101$
- $x_4=2$, $h_5(2) = 01000$
- $x_5=8$, $h_5(8) = 00100$
- $x_6=1$, $h_5(1) = 11010$

Substream	Address	Counter
S_0	00	2
S_1	01	3
S_2	10	1
S_3	11	1

LogLog algorithm

```
Input: stream S, array of m counters, hash fiction h
Output: cardinality of S

for j=0 to m-1 do:
 COUNT[j] = 0

for x in S do:
  i = h(x)
  j = getLeftBits(i, p)
  r = rank(getRightBits(i, M-p))
  COUNT[j] = max(COUNT[j], r)

  R = average(COUNT) // average of all j counters
  output a * m * 2R // a is a constant, a ≈ 0.39701, for m ≥ 64.
```

Why *LogLog*?

Let's assume we want to be able to count up to $\mathbf{n}$ distinct elements.

We need a hash function that maps each input element to $\mathbf{log_2 n}$ bits.

Then, each counter needs to be able to count up to $\mathbf{log_2(log_2 n)}$ 0s.

Combining estimates

- **Average won't work**: The expected value of 2^R is too large.
- **Median won't work**: it is always a power of 2, thus, if the correct estimate is between two powers of 2, we won't get a good estimate.

Solution: **harmonic mean** (HyperLogLog)

$$\hat{n} = a_m \cdot m^2 \cdot (\sum_{j=0}^{m-1} 2^{-COUNT[j]})$$

Standard error

The standard error of the LogLog algorithm is inversely related to the number of counters m:

$$\delta \approx \frac{1.3}{\sqrt{m}}$$

For $m = 256$, the error is about 8%

For $m = 1024$, the error decreases to 4%

Space requirements

Space requirements

As we read the stream, it is not necessary to store any elements seen:

Space requirements

As we read the stream, it is not necessary to store any elements seen:

- Assume we want to count cardinalities up to 1 billion or 2^{30} with an accuracy of 4%.

Space requirements

As we read the stream, it is not necessary to store any elements seen:

- Assume we want to count cardinalities up to 1 billion or 2^{30} with an accuracy of 4%.
- The hash value needs to map elements to $M = \log_2(2^{30}) = 30$ bits.

Space requirements

As we read the stream, it is not necessary to store any elements seen:

- Assume we want to count cardinalities up to 1 billion or 2^{30} with an accuracy of 4%.
- The hash value needs to map elements to $M = \log_2(2^{30}) = 30$ bits.
- We need 1024 counters, so $m = 2^{10}$ and we need $p = \log_2 m = 10$ bits for routing.

Space requirements

As we read the stream, it is not necessary to store any elements seen:

- Assume we want to count cardinalities up to 1 billion or 2^{30} with an accuracy of 4%.
- The hash value needs to map elements to $M = \log_2(2^{30}) = 30$ bits.
- We need 1024 counters, so $m = 2^{10}$ and we need $p = \log_2 m = 10$ bits for routing.
- Each counter needs to be able to count up to 20 0s, so we need to allocate $\log_2 20 = 4.32$ bits per counter.

Space requirements

As we read the stream, it is not necessary to store any elements seen:

- Assume we want to count cardinalities up to 1 billion or 2^{30} with an accuracy of 4%.
- The hash value needs to map elements to $M = \log_2(2^{30}) = 30$ bits.
- We need 1024 counters, so $m = 2^{10}$ and we need $p = \log_2 m = 10$ bits for routing.
- Each counter needs to be able to count up to 20 0s, so we need to allocate $\log_2 20$ = 4.32 bits per counter.
- If we round up to 5 bits, that's **640 bytes in total**.

Estimating frequencies

Motivating examples

Detect DNS DDoS attacks

- Flooding the resources of the targeted system by sending a large number of query from a botnet
- Group queries by their top-level domain and investigate most popular domains
- Alert if we detect many different non-existent subdomains of the same primary domain

Trending topics calculation

- Twitter receives around 500 billion tweets per day
- Estimating the frequencies of hashtags and comparing them with yesterday's frequencies provides an indication of what is "trending"

Counting Bloom Filter

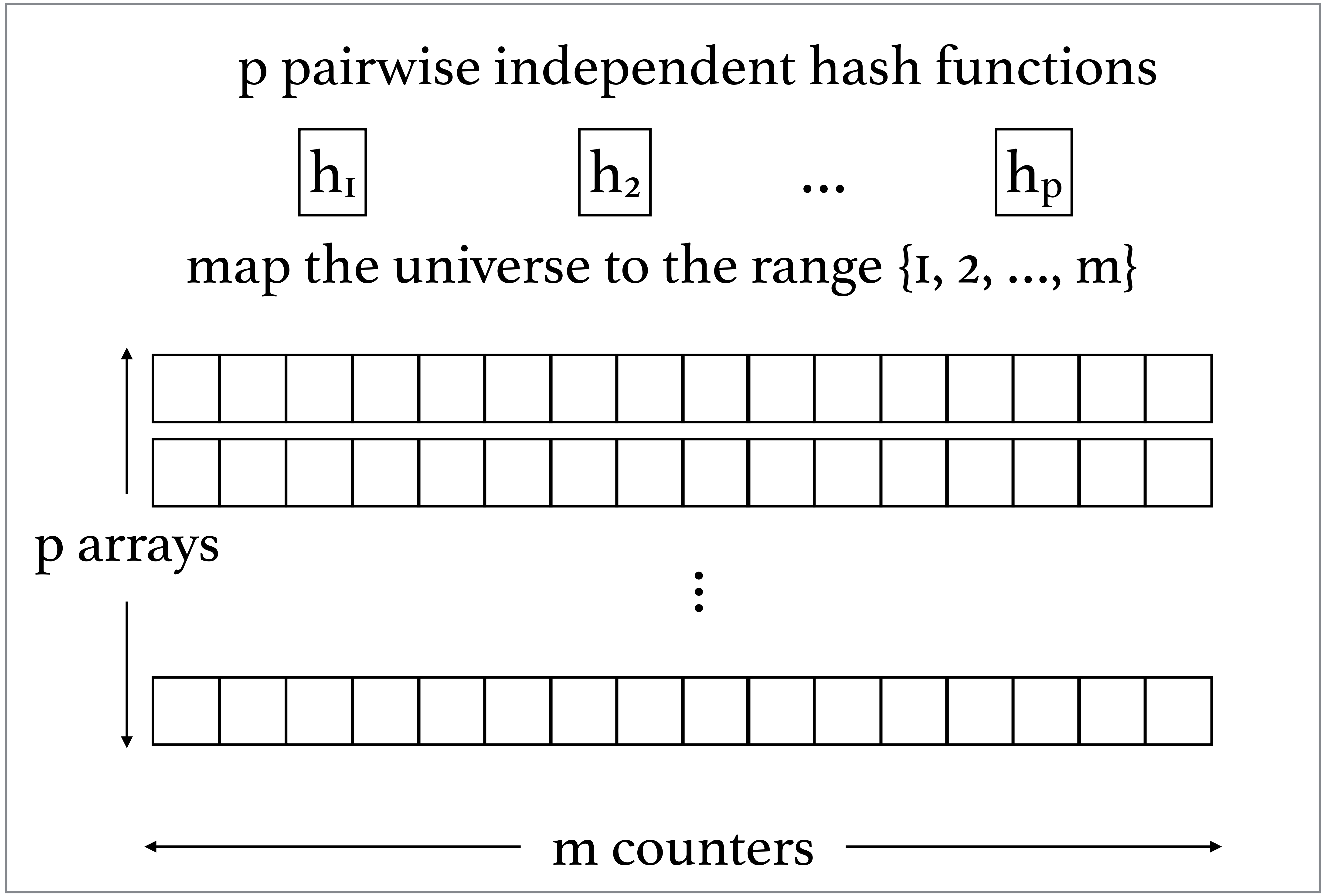
- Expand the classical BF with an array of $\mathbf{m}$ counters corresponding to each of the $\mathbf{m}$ bits in the filter:
  - Increment the corresponding counter every time an element is added
  - To delete an element, decrease its corresponding counters and unset the corresponding bit of the counter falls to 0
- A single array of counters for all hash functions increases the collision probability
- Counter overestimation is almost certain for very large data streams with high-frequency elements

The Count-Min Sketch

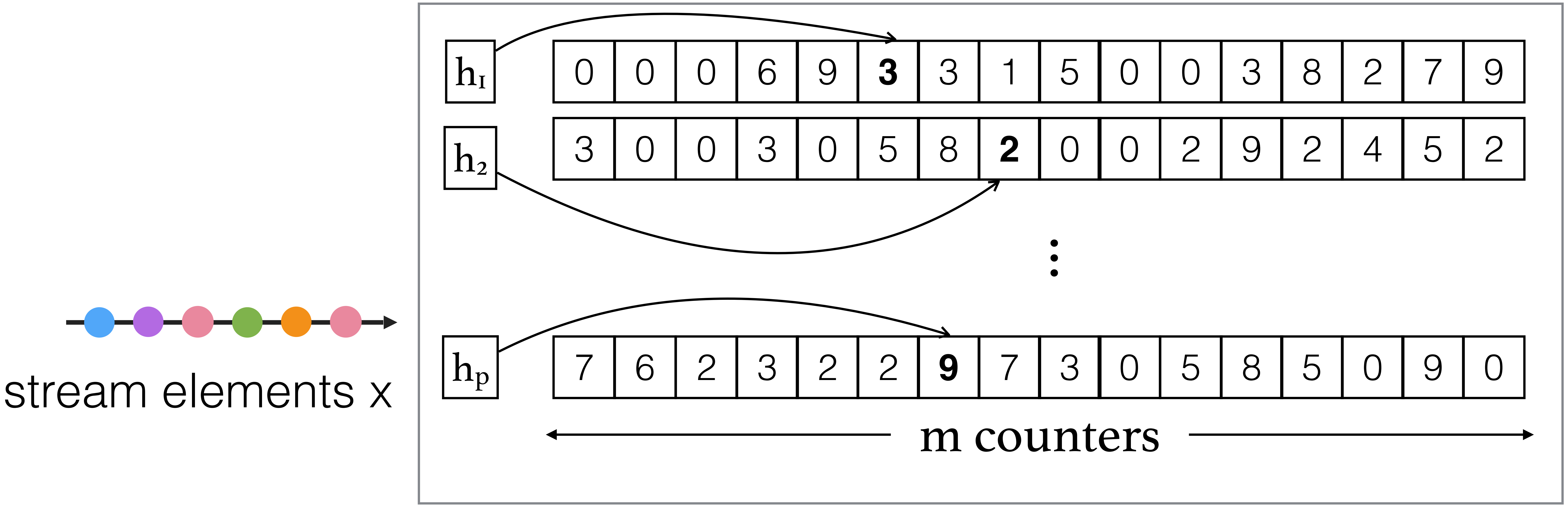
- A space-efficient probabilistic data structure that can be used to estimate frequencies and heavy hitters in data streams
- It was introduced in 2003 by Cormode and Muthukrishnan

- It uses a hash table of $\mathbf{p}$ arrays of $\mathbf{m}$ counters
- Elements update different subsets of counters, one per hash table
- Many independent trials by using $\mathbf{p}$ hash functions with an array of $\mathbf{m}$ counters for each of them

The Count-Min Sketch



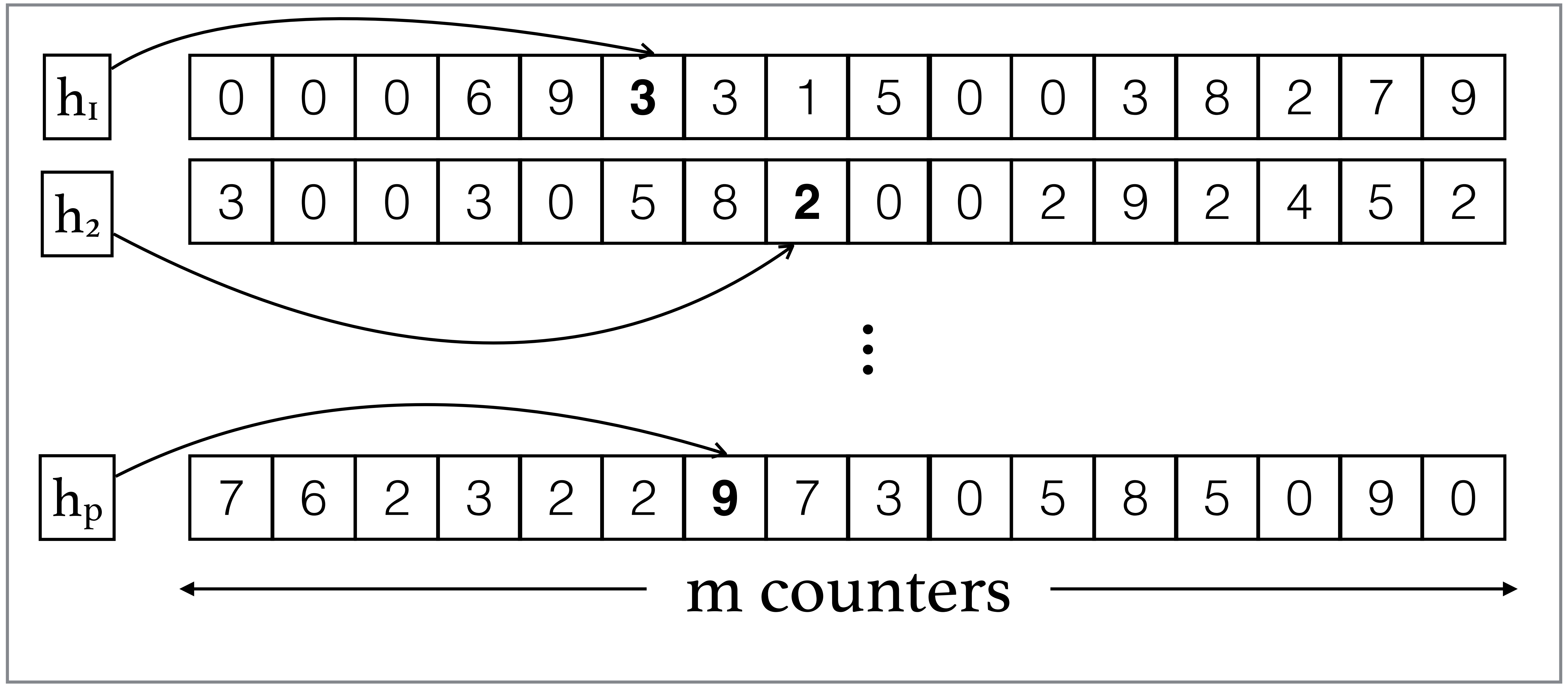
Adding an element to the sketch



All counters are initialized to 0s

```
for j=1 to p do
  i = hj(x)
  ci,j++
```

Estimating frequency




```
let f: array of length p
for j=1 to p do
  i = hj(x)
  f[j] = ci,j
return min(f[1], f[2], …, f[p])
```

Counters provide the upper bound for an element's frequency:

$$f(x) \leq c_j^{h(x)}, j = 1,2,\ldots,p$$

Because $\mathbf{m} << \mathbf{n}$, there are many collisions and counters generally overestimate real frequencies.

The best approximation is not the average of all counters, but the **minimum**.

Computing top-k

Computing top-k

- Additional to the array of counter, we allocate:
  - a counter N of the number of elements seen so far
  - a heap X^* of up to k potential heavy hitters and their frequency estimations

Computing top-k

- Additional to the array of counter, we allocate:
  - a counter N of the number of elements seen so far
  - a heap X^* of up to k potential heavy hitters and their frequency estimations
- We use a frequency threshold $f^*=N/k$ to decide whether an element is popular

Computing top-k

- Additional to the array of counter, we allocate:
    - a counter N of the number of elements seen so far
    - a heap X^* of up to k potential heavy hitters and their frequency estimations
- We use a frequency threshold $f^*=N/k$ to decide whether an element is popular
- For every element x, we add it to the sketch and then use the updated sketch to estimate its frequency.

Computing top-k

- Additional to the array of counter, we allocate:
  - a counter N of the number of elements seen so far
  - a heap X^* of up to k potential heavy hitters and their frequency estimations
- We use a frequency threshold $f^*=N/k$ to decide whether an element is popular
- For every element x, we add it to the sketch and then use the updated sketch to estimate its frequency.
- If the estimated frequency is above the threshold:
  - we add it to the heap or update its frequency if it is already in the heap

Computing top-k

- Additional to the array of counter, we allocate:
  - a counter N of the number of elements seen so far
  - a heap X^* of up to k potential heavy hitters and their frequency estimations
- We use a frequency threshold $f^*=N/k$ to decide whether an element is popular
- For every element x, we add it to the sketch and then use the updated sketch to estimate its frequency.
- If the estimated frequency is above the threshold:
  - we add it to the heap or update its frequency if it is already in the heap
- When a popular element's frequency drops below the threshold, we remove it from the heap

Computing top-k

```
N=0 // number of elements so far
X* = {} // heap of top-k elements

for x in input do:
  N = N+1
  f* = N/k // current frequency threshold
  update(x) // add x to the count-min sketch (slide 22)
  f = frequency(x) // use sketch to estimate frequency (slide 23)

  if f >= f* then:
    X*.add({x, f})
  // remove unpopular elements from the heap
  for (y, f_y) in X* do:
    if f_y <= f* then
      X*.remove({y, f_y})

return X*
```

Error and space/time trade-offs

- Query approximation error ϵ
- Error probability δ

Guarantee: The estimation error for frequencies will not exceed $\epsilon \cdot n$ with probability $1 - \delta$

- A higher number of hash functions decreases the probability of a bad estimate: $p = \lceil ln\frac{1}{\delta} \rceil$
- The recommended number of counters is $m = \lceil \frac{2.71828}{\epsilon} \rceil$

Space requirements

Space requirements

For a standard error of $\delta \approx 1\,\%$, we need at least $p = \lceil ln\frac{1}{\delta} \rceil = 5$ hash functions.

Space requirements

For a standard error of $\delta \approx 1\,\%$, we need at least $p = \lceil ln\frac{1}{\delta} \rceil = 5$ hash functions.

Consider a stream of 10 million ($n = 10^7$) elements and an allowed overestimate of 10. Thus, $\epsilon = \frac{10}{10^7} = 10^{-6}$.

Space requirements

For a standard error of $\delta \approx 1\,\%$, we need at least $p = \lceil ln\frac{1}{\delta} \rceil = 5$ hash functions.

Consider a stream of 10 million ($n = 10^7$) elements and an allowed overestimate of 10. Thus, $\epsilon = \frac{10}{10^7} = 10^{-6}$.

The recommended number of counters is $m = \frac{2.71828}{10^{-6}} \approx 2{,}718{,}280$.

Space requirements

For a standard error of $\delta \approx 1\,\%$, we need at least $p = \lceil ln\frac{1}{\delta} \rceil = 5$ hash functions.

Consider a stream of 10 million ($n = 10^7$) elements and an allowed overestimate of 10. Thus, $\epsilon = \frac{10}{10^7} = 10^{-6}$.

The recommended number of counters is $m = \frac{2.71828}{10^{-6}} \approx 2{,}718{,}280$.

The sketch data structure requires a counter array of size 5 * 2,718,280.

Space requirements

For a standard error of $\delta \approx 1\,\%$, we need at least $p = \lceil ln\frac{1}{\delta} \rceil = 5$ hash functions.

Consider a stream of 10 million ($n = 10^7$) elements and an allowed overestimate of 10. Thus, $\epsilon = \frac{10}{10^7} = 10^{-6}$.

The recommended number of counters is $m = \frac{2.71828}{10^{-6}} \approx 2{,}718{,}280$.

The sketch data structure requires a counter array of size 5 * 2,718,280.

Considering 32-bit counters, the count-min sketch requires a total of **54.4MB** of memory.

Further reading

- Jure Lescovec, Anand Rajaraman and Jeffrey David Ullman. **Mining of Massive Datasets**. http://infolab.stanford.edu/~ullman/mmds/book.pdf
- Durand, Marianne, and Philippe **Flajolet. Loglog counting of large cardinalities**. *European Symposium on Algorithms,* 2003.
- Flajolet, Philippe, et al. **Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm**. 2007. https://hal.archives-ouvertes.fr/file/index/docid/406166/filename/FlFuGaMe07.pdf
- Cormode, Graham, and Shan Muthukrishnan. **An improved data stream summary: the count-min sketch and its applications**. *Journal of Algorithms* (2005).
- Gakhov, Andrii. **Probabilistic Data Structures and Algorithms for Big Data Applications**. 2019.