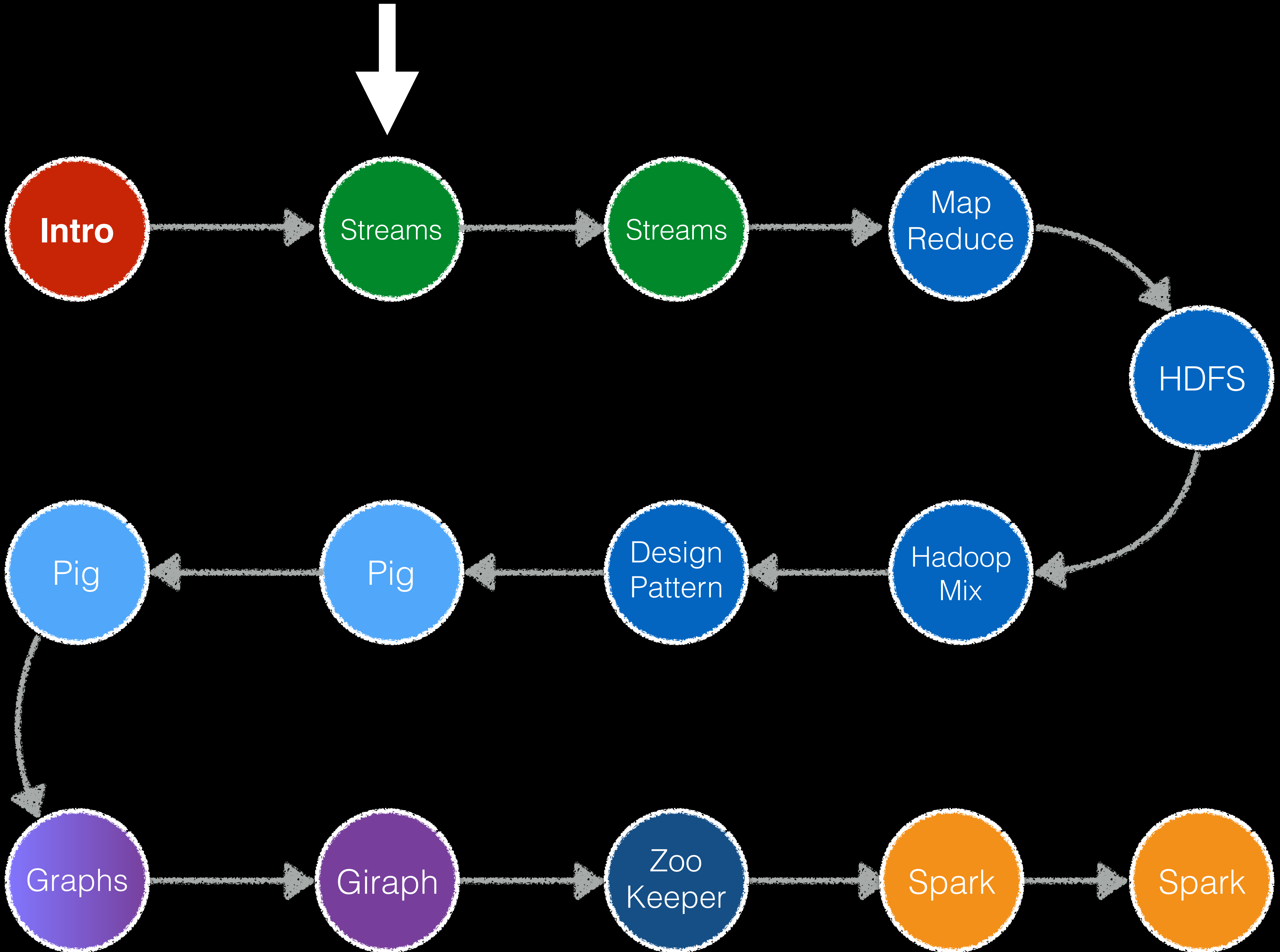
TI2736-B

Big Data Processing

Claudia Hauff

ti2736b-ewi@tudelft.nl

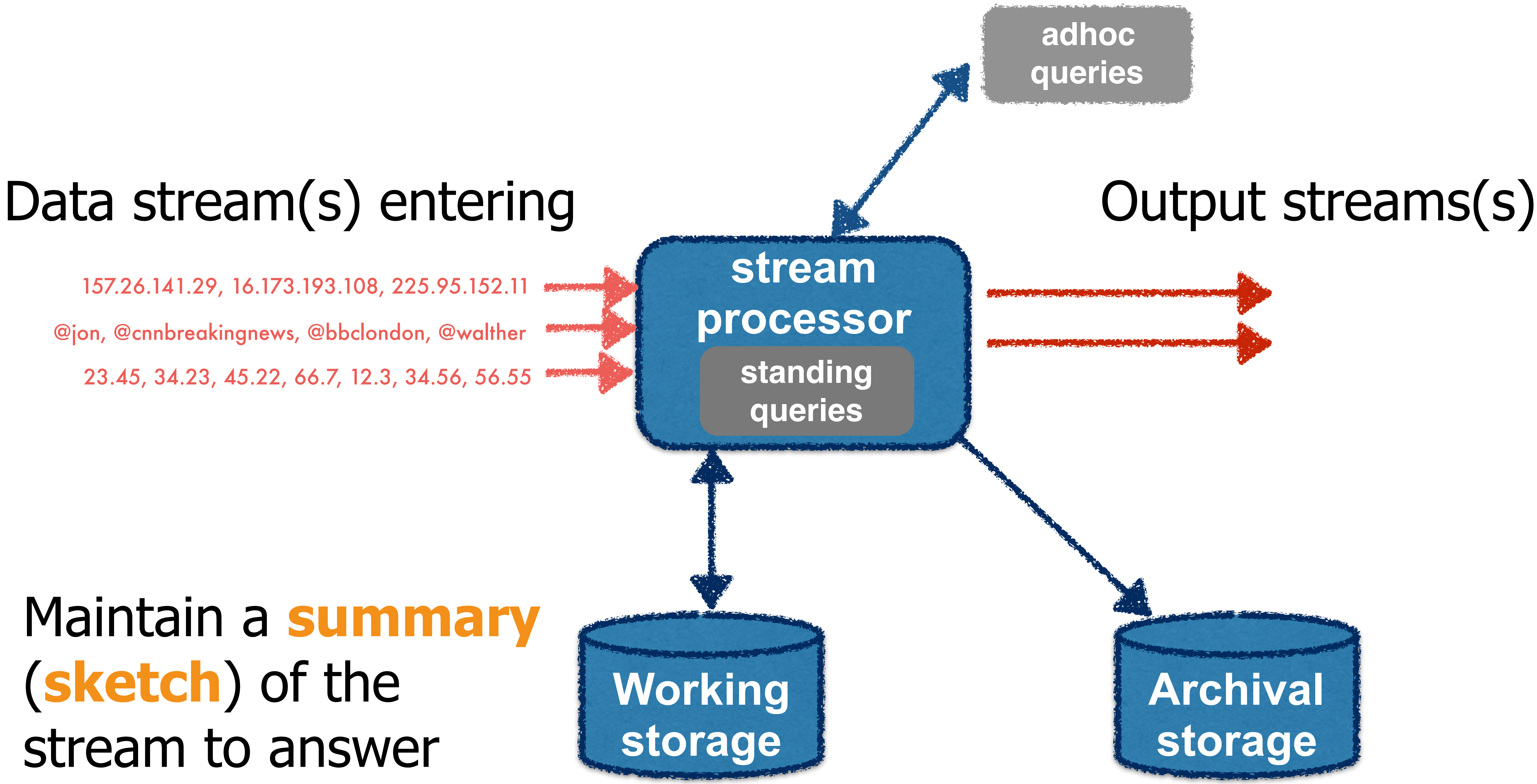

Intro
Streams
Streams
Map Reduce
HDFS
Hadoop Mix
Design Pattern
Pig
Pig
Graphs
Giraph
Zoo Keeper
Spark
Spark

Learning objectives

- **Explain** the limiting factors of data streaming & describe the different data stream models
- **Implement** sampling approaches for data streams
  - RESERVOIR sampling
  - MIN-WISE sampling
- **Implement** counter-based frequent item estimation approaches
  - MAJORITY
  - FREQUENT
  - SPACE-SAVING
- **Implement** BLOOM filters

Data streaming

Streaming architecture



Maintain a **summary** (**sketch**) of the stream to answer **queries**.

Data streaming scenario

- **Continuous** and rapid input of data
- **Limited memory** to store the data (less than linear in the input size)
- **Limited time** to process each element
- **Sequential** access (no random access)
- Algorithms have **one** ($p=1$) or very **few passes** ($p=\{2,3\}$) over the data

Data streaming scenario

- Typically: **simple functions** of the stream are computed and used as input to other algorithms
  - Number of *distinct* items
  - Heavy hitters
  - ….

- Closed form solutions are rare - **approximation** and **randomisation** are the norm

Data stream models

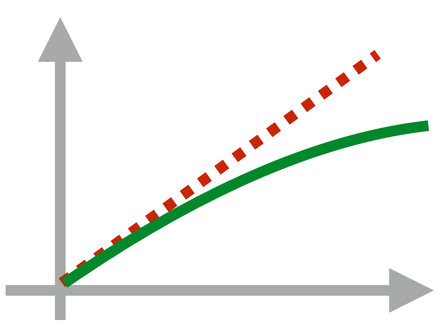
- **Massively** long input stream

stream length *m*

not a restriction: requires a single preprocessing step to convert symbols to integers

- **Basic "vanilla" model**:

$$\sigma =< a_1, a_2, a_3, .., a_m >$$



$$with\ elements\ drawn\ from\ [n] := 1, 2, ..., n$$

universe size *n*

- **Space complexity goal**: s bits of random-access memory with $s = o(min\{m, n\})$

$s = O(\log m + \log n)$
"holy grail"

$s = poly\log(min(m, n))$
"reality"

Data stream models

- **Frequency vectors**: computing some statistical property from the multi-set of items in the input stream

$$\mathbf{f} = (f_1, f_2, ..., f_n) \ \ where \ f_j = |i : a_i = j|$$

$$with \ \mathbf{f} \ starting \ at \ 0$$

- **Turnstile model**: elements can "arrive" and "depart" from the multi-set by variable amounts

$$upon \ receiving \ a_i = (j, c), update \ f_j \leftarrow f_j + c$$

- **Cash register model**: only positive updates are allowed

Data stream models

- **Frequency vectors**: computing some statistical property from the multi-set of items in the input stream

$$\mathbf{f} = (f_1, f_2, ..., f_n) \ \textit{where} \ f_j = |i : a_i = j|$$

$$\textit{with} \ \mathbf{f} \ \textit{starting at} \ 0$$

- **Turnstile model**: elements can "arrive" and "depart" from the multi-set by variable amounts

$$\textit{upon receiving} \ a_i = (j, c), \textit{update} \ f_j \leftarrow f_j + c$$

A data streaming algorithm A takes the stream as input and computes a function $\phi(\sigma)$

Data stream models

"For instance, estimating cardinalities [**number of distinct elements**] … of **a hundred million different records** can be achieved with m=2048 memory units of 5 bits each, which corresponds to **1.28 kilobytes of auxiliary storage** in total, the **error** observed being typically **less than 2.5%**."

Durand, Marianne, and Philippe Flajolet. "Loglog counting of large cardinalities." Algorithms-ESA 2003. Springer Berlin Heidelberg, 2003. 605-617.

Data stream models

- **Frequency vectors**: computing some statistical

"The best methods can be implemented to find **frequent items** with high accuracy using only **tens of kilobytes of memory**, at rates of **millions of items per second** on **cheap modern hardware**."

Cormode, Graham, and Marios Hadjieleftheriou. "Finding frequent items in data streams." Proceedings of the VLDB Endowment 1.2 (2008): 1530-1541.

and computes a function $\phi(\sigma)$

Data stream models

“consider the problem of deriving an **execution plan** for a **query** expressed in a declarative language such as SQL. There usually exist **several alternative plans** that all produce the same result, but they can differ in their efficiency by **several orders of magnitude**”

Gemulla, Rainer. "Sampling algorithms for evolving datasets." (2008).

Data stream models

"The main idea behind this processing model [**approximate query processing**] is that the **computational cost** of query processing can be reduced when the underlying application **does not require exact results** but only a highly-accurate estimate thereof"

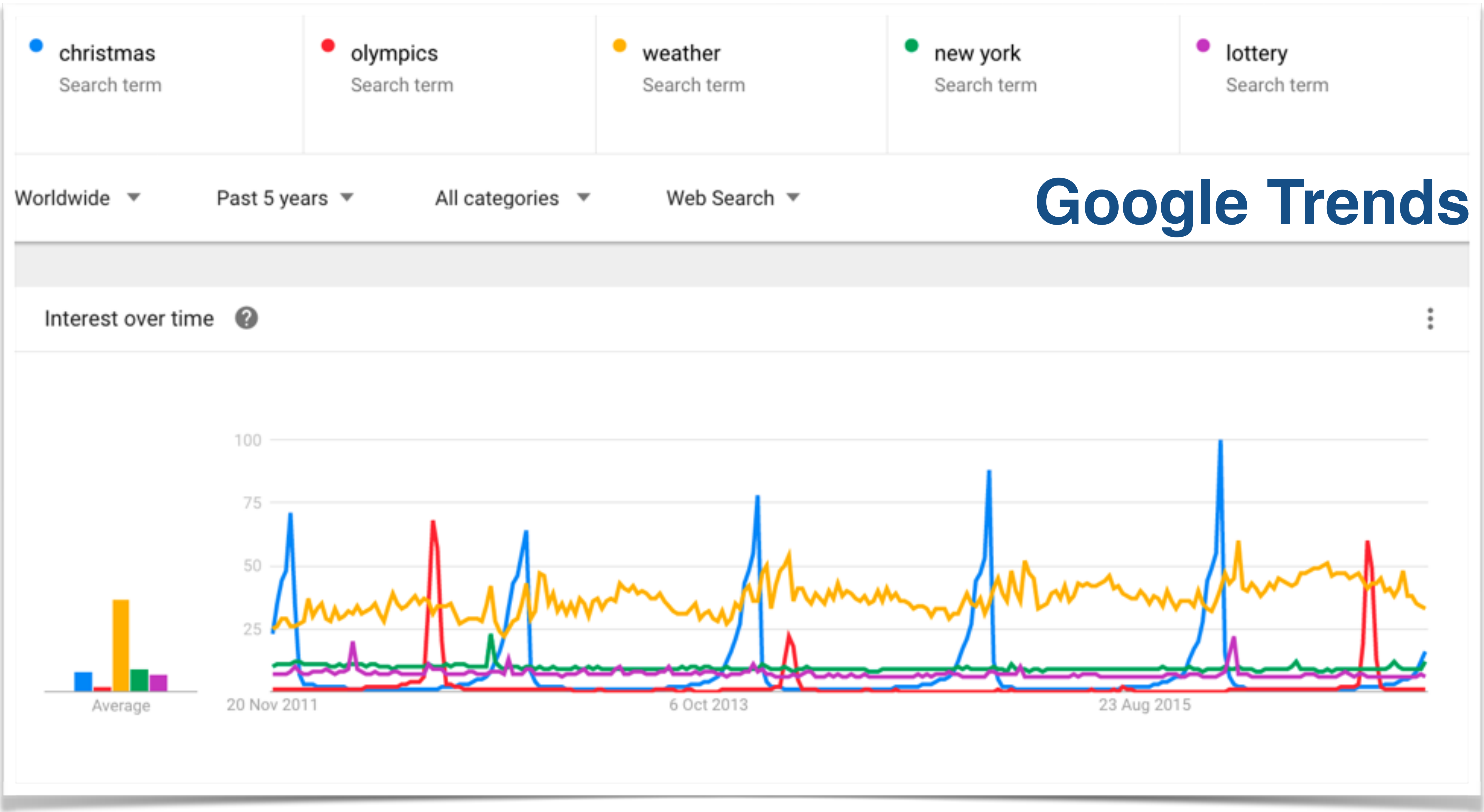
Gemulla, Rainer. "Sampling algorithms for evolving datasets." (2008).

Sampling

Overview

- Sampling: selection of a **subset of items** from a large data set
- Goal: sample **retains the properties of the whole** data set
- Important for drawing the right conclusions from the data

Overview



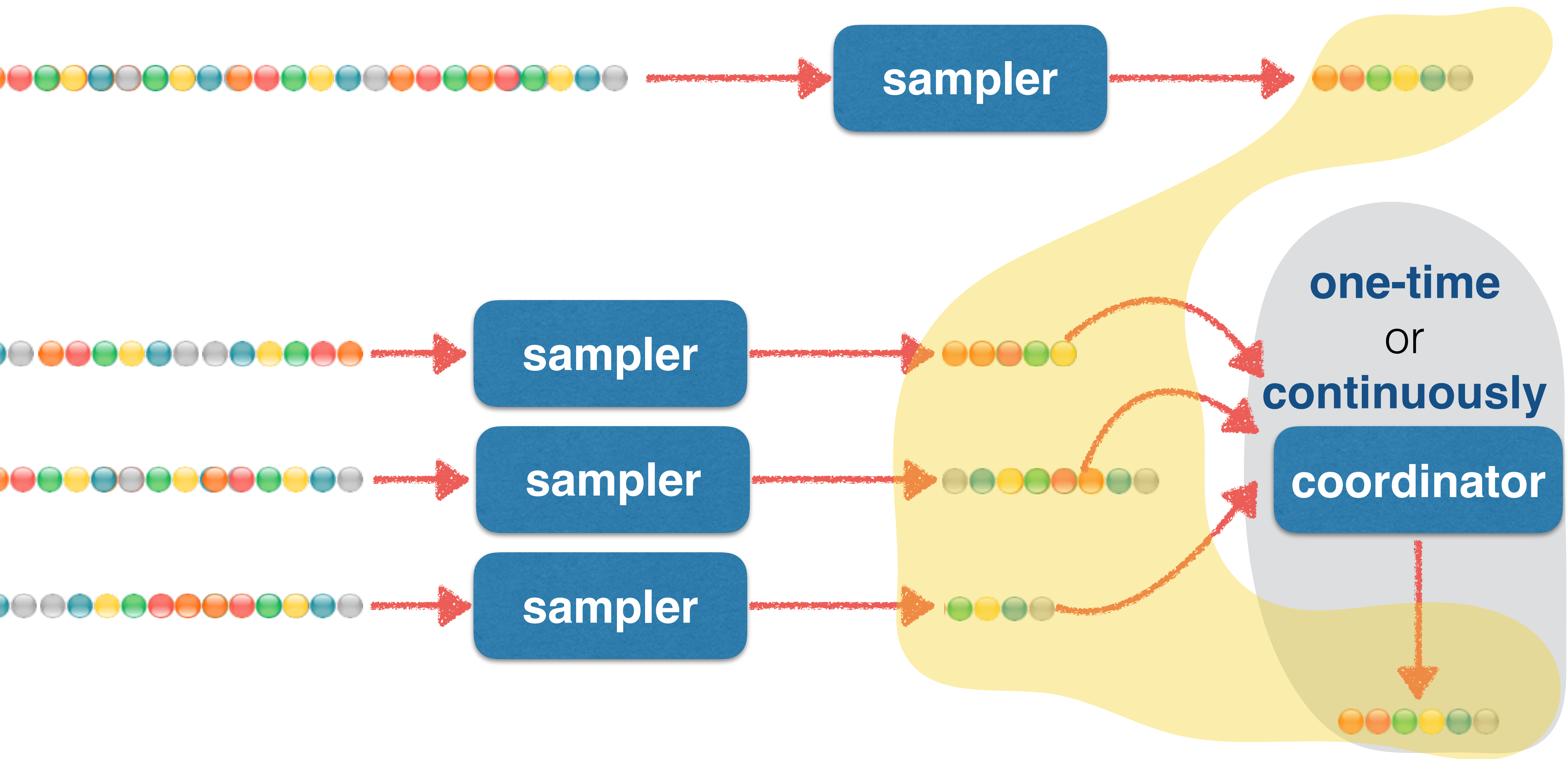
christmas
Search term
olympics
Search term
weather
Search term
new york
Search term
lottery
Search term
Worldwide
Past 5 years
All categories
Web Search
Google Trends
Interest over time
100
75
50
25
Average
20 Nov 2011
6 Oct 2013
23 Aug 2015

Sampling framework

- Algorithm A **chooses** every incoming element **with a certain probability**
- If the element is **sampled**, A puts it into memory, otherwise the element is **discarded**
- Algorithm A may discard some items from memory after having added them
- For every query, A computes some function $\phi(\sigma)$ **only based on the in-memory sample**

Single machine vs. distributed

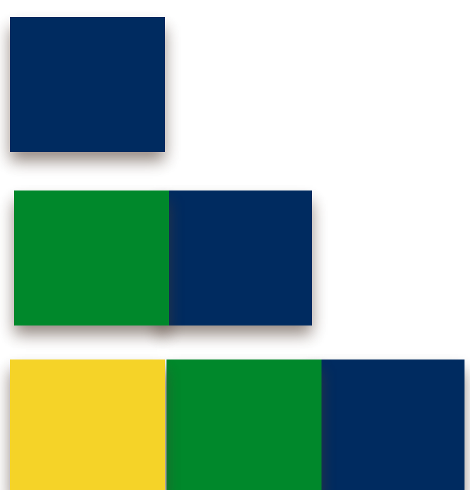
at **any** point in time, the sample should be valid



Reservoir sampling

a reservoir of valid random samples

Task: Given a data stream of **unknown length**, randomly pick k elements from the stream so that **each** element has the **same probability** of being chosen.



m=1 keep it

m=2 replace with probability 1/2

m=3 replace / with probability 1/3

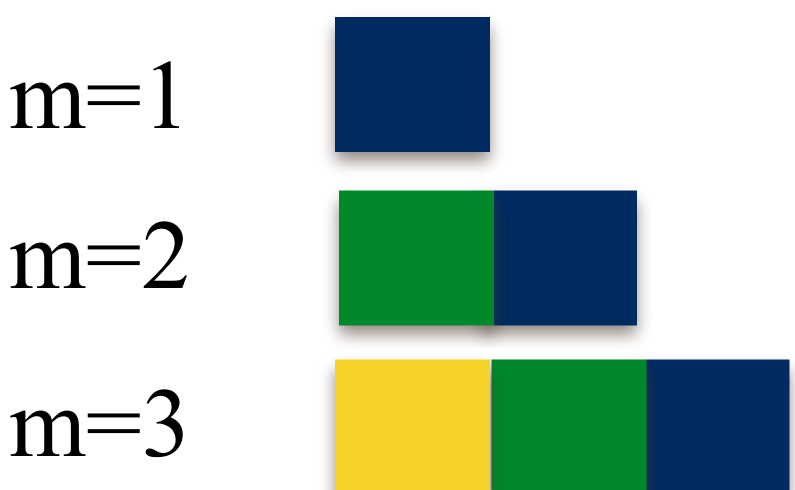
keep / with probability 2/3

Toy example with k=1

Reservoir sampling

a reservoir of valid random samples

Task: Given a data stream of **unknown length**, randomly pick $\boldsymbol{k}$ elements from the stream so that **each** element has the **same probability** of being chosen.



Toy example with k=1

$$P({\color{navy}\blacksquare}) = 1 \times \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$$

$$P({\color{green}\blacksquare}) = \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$$

$$P({\color{gold}\blacksquare}) = \frac{1}{3}$$

Reservoir sampling

sampling without replacement

(1) Sample the first $\boldsymbol{k}$ elements from the stream

(2) Sample the $\boldsymbol{i^{th}}$ element ($i>k$) with probability $\boldsymbol{k/i}$ (if sampled, randomly replace a previously sampled item)

- **Limitations**:
  - Wanted sample has to fit into main memory
  - Distributed sampling is not trivial

Reservoir sampling example

- Stream of numbers with a **normal distribution** *N(0,1)*

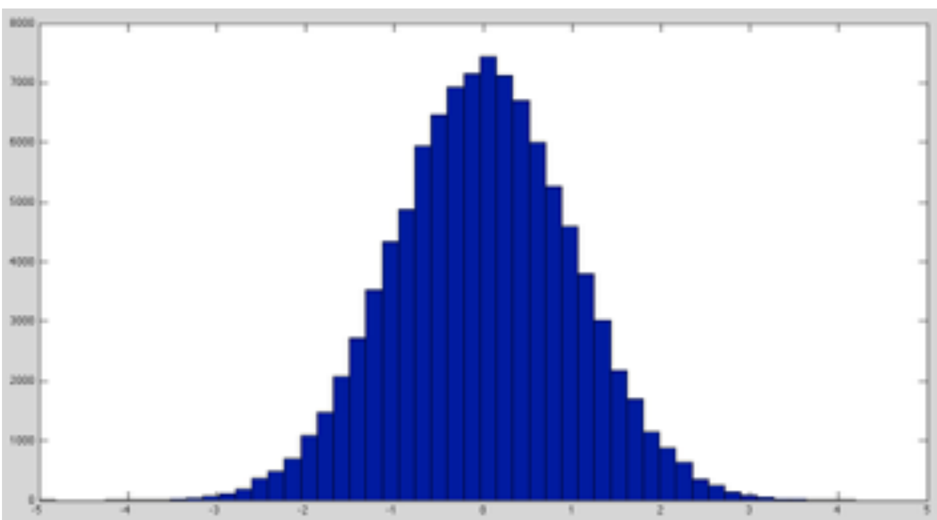
$$|S| = 100000$$

$$k = \{100, 500, 1000, 10000\}$$

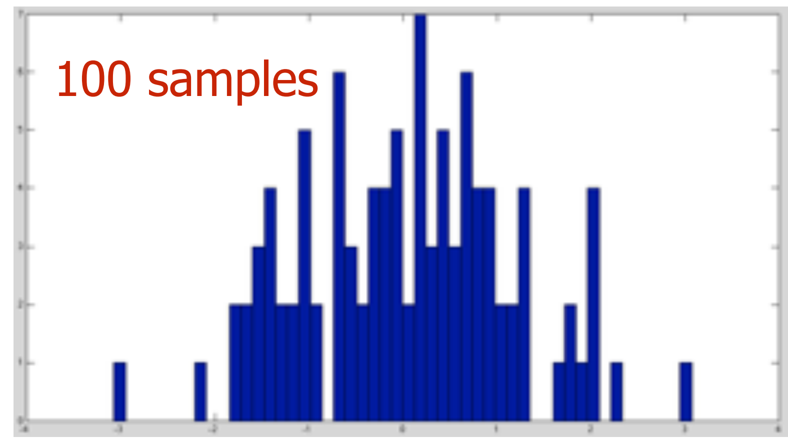
- Samples are plotted in histogram form

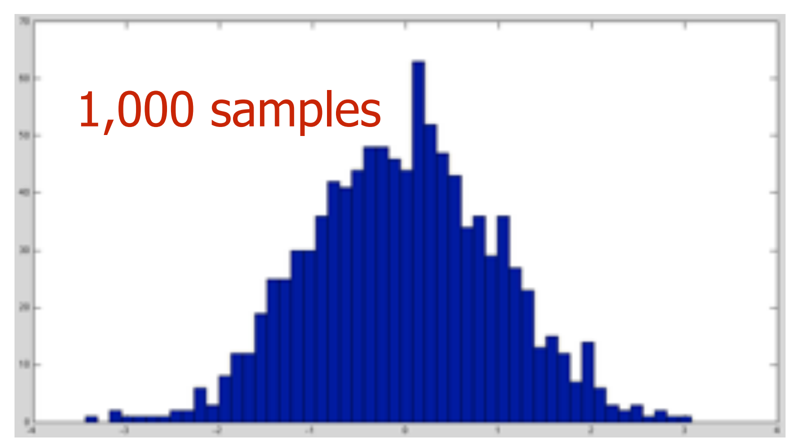
- **Expectation**: with larger $\boldsymbol{k}$, the histograms become more similar to the full stream histogram

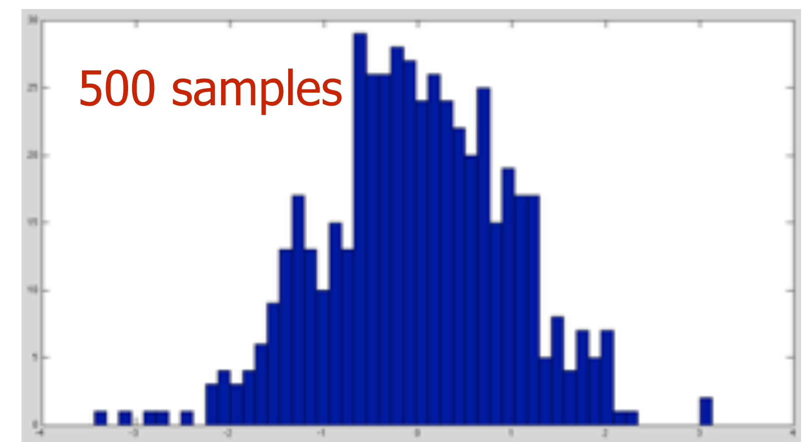
Reservoir sampling example

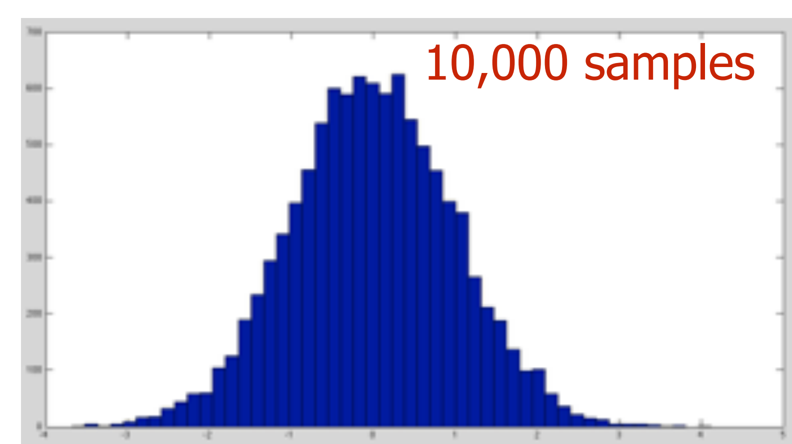


Histogram of entire stream (100,000 items)









Distributed reservoir sampling for one-time sampling

reservoir sampling sub-stream S_1

length m_1

reservoir sampling sub-stream S_2

length m_2

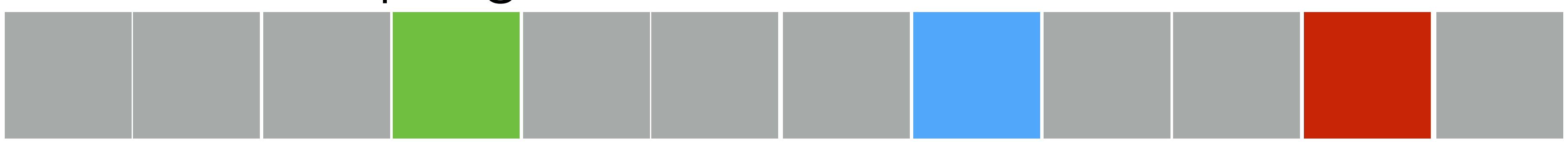
Goal: sample sub-streams in parallel, combine with the same guarantee as the non-distributed version.

Sub-stream output: k samples and length of sub-stream

Distributed reservoir sampling for one-time sampling

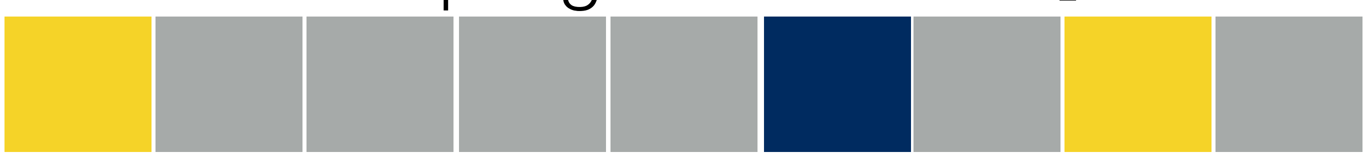
k=3

reservoir sampling sub-stream S_1



length m_1

reservoir sampling sub-stream S_2



length m_2

Combining sub-stream pairs in 2. sampling phase

k iterations:

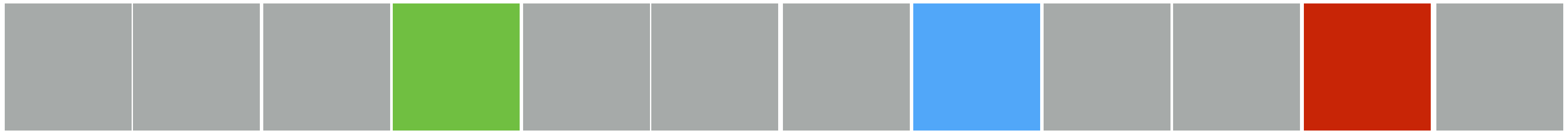
- with probability $p = \dfrac{m_1}{m_1 + m_2}$ pick a sample from S_1,
- with $(1-p)$ pick a sample from S_2

Distributed reservoir sampling for one-time sampling

k=3

not feasible for continous maintenance of distributed stream

reservoir sampling sub-stream S_1



length m_1

reservoir sampling sub-stream S_2

length m_2

Combining sub-stream pairs in 2. sampling phase

k iterations:

- with probability $p = \dfrac{m_1}{m_1 + m_2}$ pick a sample from S_1,
- with $(1 - p)$ pick a sample from S_2

Min-wise sampling

Task: Given a data stream of **unknown length**, randomly pick k elements from the stream so that **each** element has the **same probability** of being chosen.

1. For each element in the stream, tag it with a random number in the interval [0,1].
2. Keep the k elements with the smallest random tags.

Min-wise sampling

Task: Given a data stream of **unknown length**, randomly pick k elements from the stream so that **each** element has the **same probability** of being chosen.

- Can easily be run in a **distributed** fashion with a merging stage (every subset has the same chance of having the smallest tags)
- Disadvantage: **more memory/CPU intensive** than reservoir sampling ("tags" need to be stored as well)

Sampling: summary

- **Advantages**:
  - Low cost
  - Efficient data storage
  - Classic algorithms can be run on it (all samples should fit into main memory)

- In practical applications, we have complicating factors:
  - **Time-sensitive window**: only the last x items of the stream are of interest (e.g. in anomaly detection)
  - **Sampling from databases** through their indices from **non-cooperative** providers (e.g. Google, Bing)
    - How many car repairs does Google Places index?
    - How many documents does Google index?

Frequency counter algorithms

"Counter-based algorithms track a **subset** of items from the inputs, and **monitor counts** associated with these items.
For **each new arrival**, the algorithms decide **whether to store this item or not**, and if so, what counts to associate with it."

Examples

Packets on the Internet

Frequent items: **most popular destinations** or **most heavy bandwidth users**

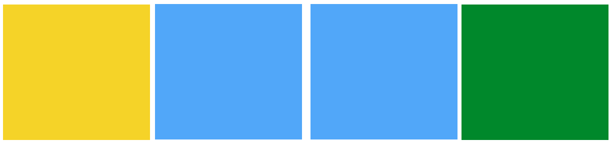
Queries submitted to a search engine

Frequent items: **most popular queries**

MAJORITY algorithm

Task: Given a list of elements - is there an **absolute majority** (an element occurring $> \frac{m}{2}$ times)?

no absolute majority



blue wins

```
c ← 0; v unassigned;
for each i :
    if c = 0 :
        v ← i;
        c ← 1;
    else if v = i :
        c ← c+1;
    else:
        c ← c-1;
```

MAJORITY algorithm

Task: Given a list of elements - is there an **absolute majority** (an element occurring $> \frac{m}{2}$ times)?

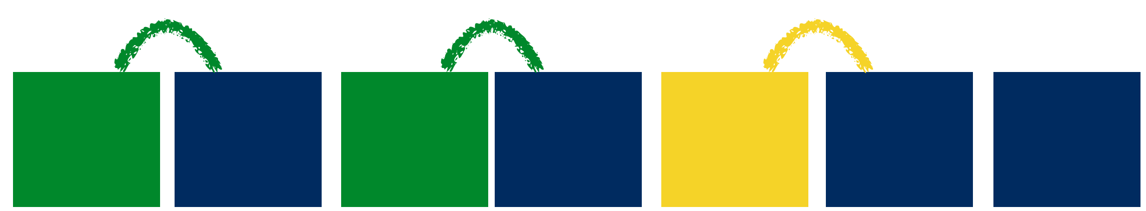
In this stream, the last item is kept.

v		b	b	b	b	b	b	b
c	0	1	0	1	2	1	0	1

A **second pass** is needed to verify if the stored item is indeed the absolute majority item (count every occurrence of $\boldsymbol{b}$).

MAJORITY algorithm

Task: Given a list of elements - is there an **absolute majority** (an element occurring $> \frac{m}{2}$ times)?



v		g	g	g	g	y	y	b
c	0	1	0	1	0	1	0	1

Correctness based on pairing argument:

- Every **non-majority element** can be paired with a majority one
- After the pairing, there will still be **majority elements** left

FREQUENT algorithm (Misra-Gries)

Task: Find all elements in a sequence whose frequency exceeds $\frac{1}{k}$ fraction of the total count (i.e. frequency $> \frac{m}{k}$)

- Wanted: **no false negatives**, i.e. all elements with frequency $> \frac{m}{k}$ need to be reported
- **Deterministic** approach

$c[1,..(k-1)] = 0; T \leftarrow \varnothing;$
for each i :
  if $i \in T$:
    $c_i \leftarrow c_i + 1;$
  else if $|T| < k-1$:
    $T \leftarrow T \cup \{i\};$
    $c_i \leftarrow 1;$
  else for all $j \in T$:
    $c_j \leftarrow c_j - 1;$
    if $c_j = 0$:
      $T \leftarrow T \setminus \{j\};$

(k-1) counter-value pairs

FREQUENT algorithm (Misra-Gries)

Blue and green have been estimated to each occur 3 times.

$k = 3$

$c = 0$

v_1	g	g	g	g	g	g	g	g	g	g	g	g
c_1	1	2	2	3	3	2	1	1	2	2	3	3
v_2	-	-	b	b	b	b	-	b	b	b	b	b
c_2	0	0	1	1	2	1	0	1	1	2	2	3

Stream with $m = 12$ elements; all elements with more than $\frac{m}{k}$ (i.e. $12/3 = 4$) occurrences should be reported.

FREQUENT algorithm (Misra-Gries)

Green is estimated to have occurred once.

$k = 3$

$c = 0$

v_1	g	g	g	g	g	g	g
c_1	1	2	2	3	3	2	1
v_2	-	-	b	b	b	b	-
c_2	0	0	1	1	2	1	0

Stream with $m = 7$ elements; all elements with more than $\frac{m}{k}$ (i.e. $7/3 = 2.333$) occurrences should be reported.

FREQUENT algorithm (Misra-Gries)

$k = 3$

$c = 0$

v_1	g	g	g	g
c_1	1	2	2	3
v_2	-	-	b	b
c_2	0	0	1	1

Recall: no false negatives wanted; blue is a **false positive** (possible, not as undesired as a false negative)

Streaming algorithms are **approximations** (estimates) of the correct answers!

Stream with $m = 4$ elements; all elements with more than $\frac{m}{k}$ (i.e. $4/3 = 1.333$) occurrences should be reported.

FREQUENT algorithm (Misra-Gries)

space complexity

- Implementation: associative array using a balanced binary search tree
- Each key has a max. value of n, each counter has a max. value of m
- At most $(k\text{-}1)$ key/counter pairs in memory at any time

$$s = O(k(\log m + \log n))$$

FREQUENT algorithm (Misra-Gries)

answer quality of frequency estimates

Counter c_j is incremented only when j occurs, thus $\hat{f}_j \leq f_j$

When c_j is decremented, $(k-1)$ counters are decremented overall (all distinct tokens); for a stream of size m, there can be at most $\frac{m}{k}$ decrements, thus:

$$f_j - \frac{m}{k} \leq \hat{f}_j \leq f_j$$

$c[1,..(k-1)] = 0; T \leftarrow \varnothing;$
for each i **:**
  if $i \in T$ **:**
    $c_i \leftarrow c_i + 1;$
  else if $|T| < k-1$ **:**
    $T \leftarrow T \cup \{i\};$
    $c_i \leftarrow 1;$
  else for all $j \in T$**:**
    $c_j \leftarrow c_j - 1;$
    if $c_j = 0$ **:**
      $T \leftarrow T \setminus \{j\};$

FREQUENT algorithm (SPACE-SAVING)

Task: Find all elements in a sequence whose frequency exceeds $\frac{1}{k}$ fraction of the total count (i.e. frequency $> \frac{m}{k}$)

- Counters are **not reset**, the element with minimum count is simply replaced
- Maximum **overestimation** can be tracked

$c[1,..(k-1)] = 0; T \leftarrow \varnothing;$

for each i :

    if $i \in T$:

        $c_i \leftarrow c_i + 1;$

    else if $|T| < k - 1$:

        $T \leftarrow T \cup \{i\};$

        $c_i \leftarrow 1;$

    else :

        $j \leftarrow \arg\min_{j \in T} c_j;$

        $c_i \leftarrow c_j + 1;$

        $T \leftarrow T \cup \{i\} \setminus \{j\};$

Experiments

- Datasets
  - Synthetic data
  - 24 hours of HTTP/UDP traffic from a backbone router in a large network
- Goal: track most frequent IP addresses

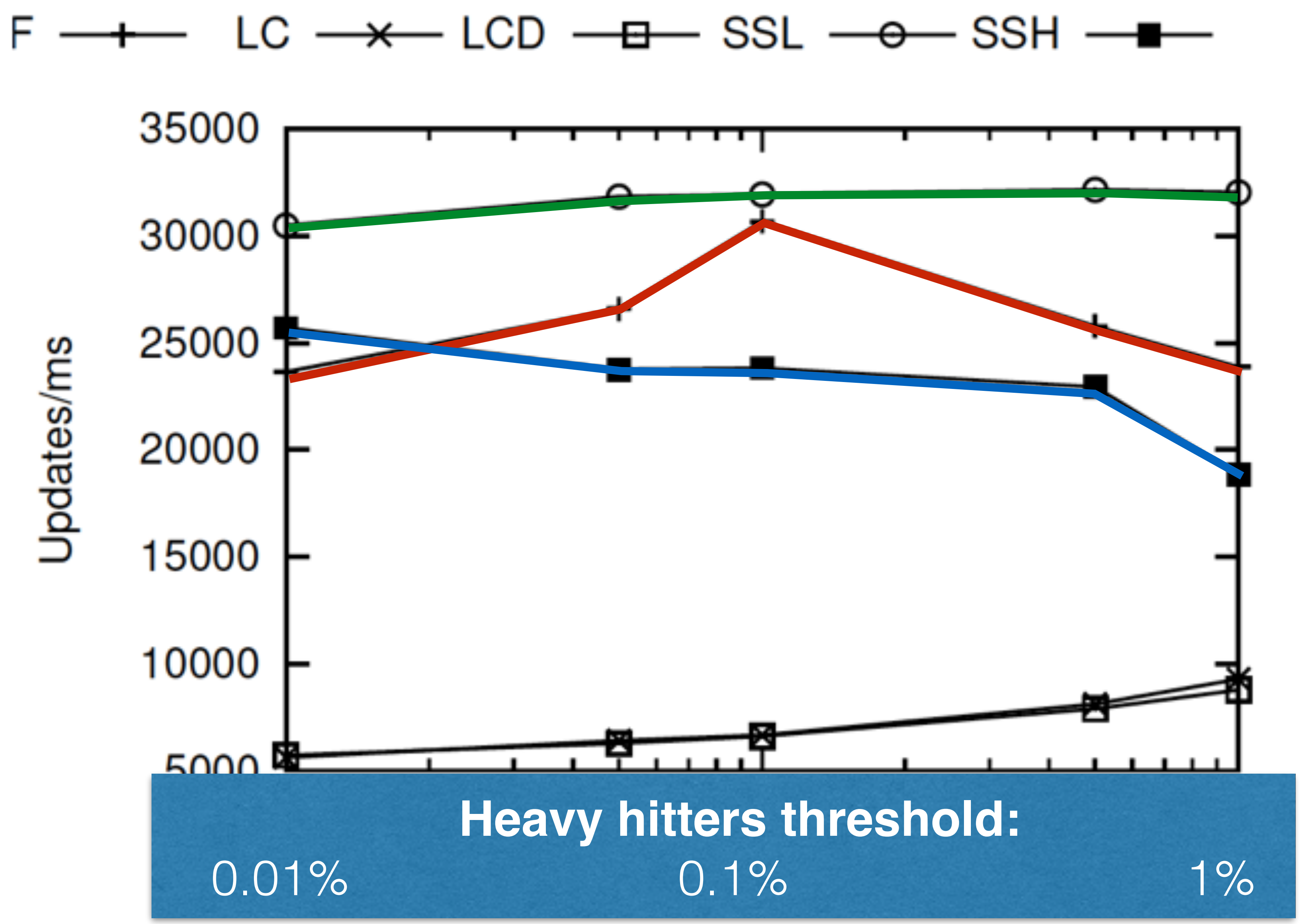
Cormode, Graham, and Marios Hadjieleftheriou. "Finding frequent items in data streams." Proceedings of the VLDB Endowment 1.2 (2008): 1530-1541.

Experiments

FREQUENT

SPACESAVING-LinkedList

SPACESAVING-Heap



(d) UDP: Speed vs. ϕ.

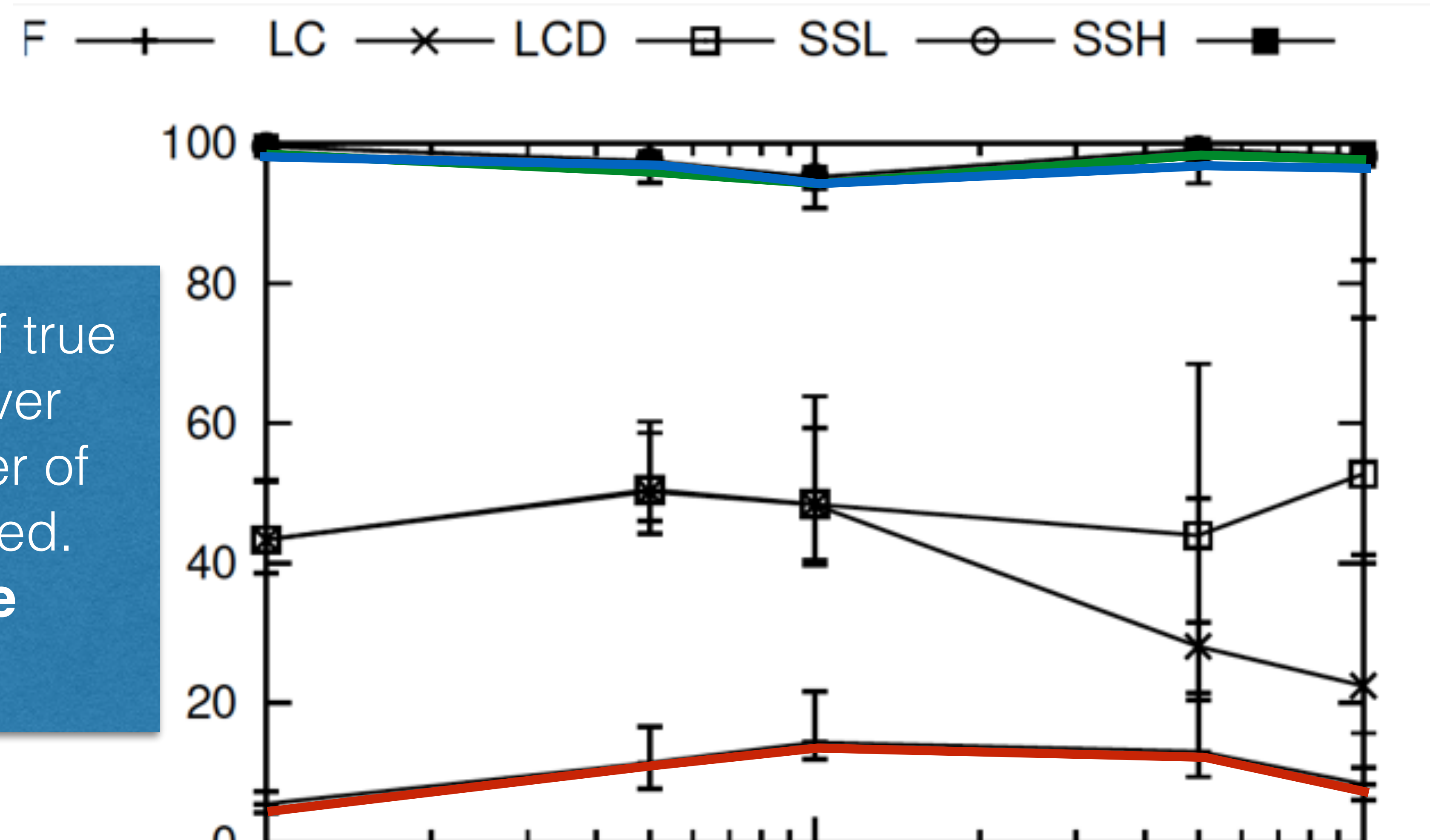
Experiments

FREQUENT

SPACESAVING-LinkedList

SPACESAVING-Heap

Total number of true heavy hitters over the total number of answers reported. Quantifies **false positives**.



Heavy hitters threshold:

0.01% 0.1% 1%

(e) UDP: Precision vs. ϕ.

Experiments

"Overall, the SPACESAVING algorithm appears **conclusively better** than other counter-based algorithms, **across a wide range of data types and parameters**. Of the two implementations compared, **SSH** exhibits very good performance in practice. It yields **very good estimates** […] consumes **very small space** and is fairly fast to update."

Cormode, Graham, and Marios Hadjieleftheriou. "Finding frequent items in data streams." Proceedings of the VLDB Endowment 1.2 (2008): 1530-1541.

Filtering

Summarizing vs. filtering

- **So far: all data is useful**, summarise for lack of space/time
- **Now**: **not all data is useful**, some is harmful
- Classic example: **spam filtering**
  - Mail servers can analyse the textual content
  - Mail servers have blacklists
  - Mail servers have whitelists (very effective!)
  - Incoming mails form a stream; quick decisions needed (delete or forward)
- Applications in Web caching, packet routing …

Problem statement

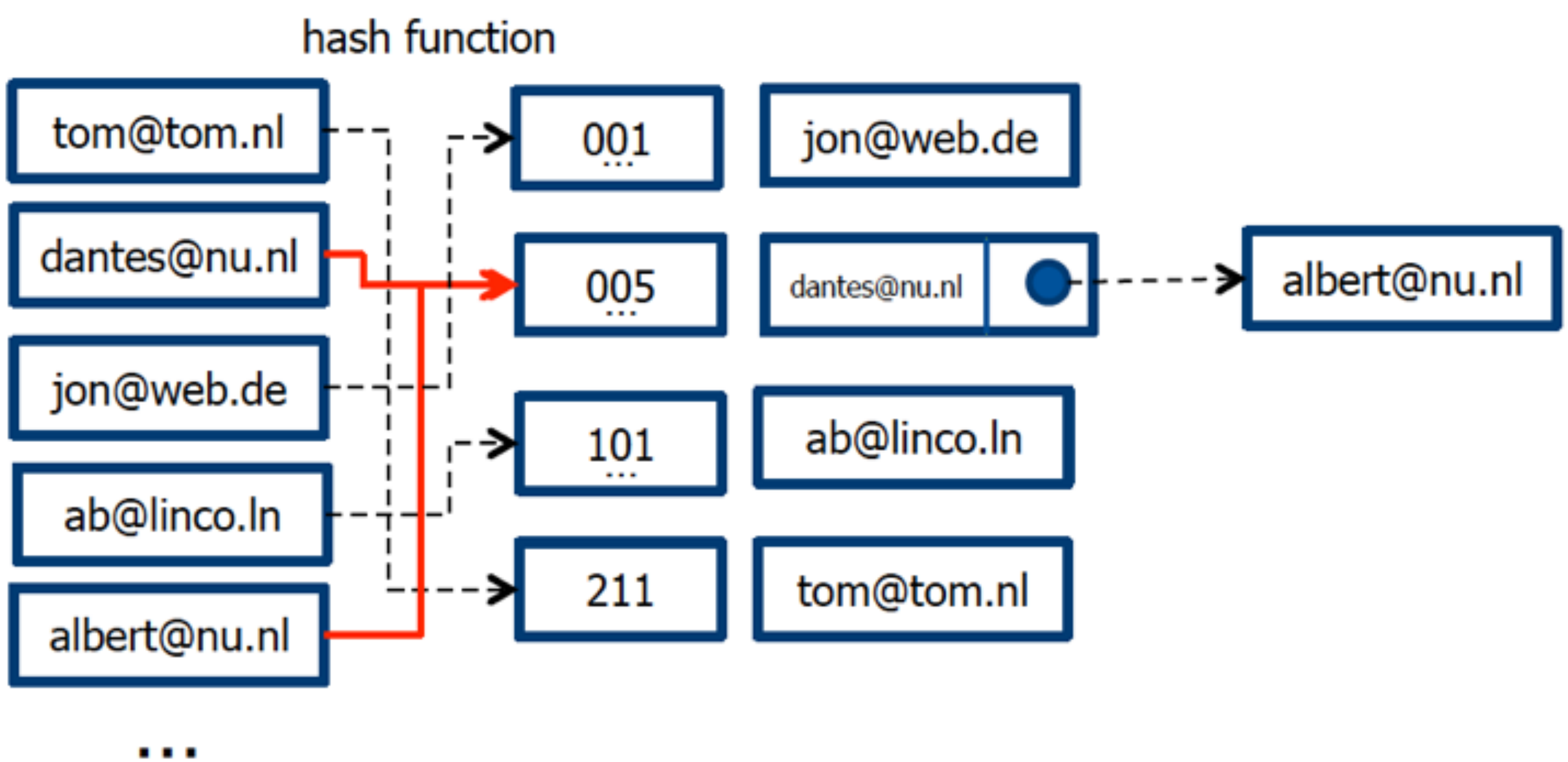
- A set W containing m values (e.g. IP addresses, email addresses, etc.)

- **Working memory of size n bit**

- **Goal**: data structure that allows **fast** checking whether the next element in the stream is in W
  - return `TRUE` **with probability 1** if the element is indeed in W
  - return `FALSE` **with high probability** if the element is not in W

A reminder: hash functions

Each element is hashed into an integer (avoid hash collisions if possible)



Bloom filter

Hash function maps each item in the universe to a **random** number **uniform** over the range.

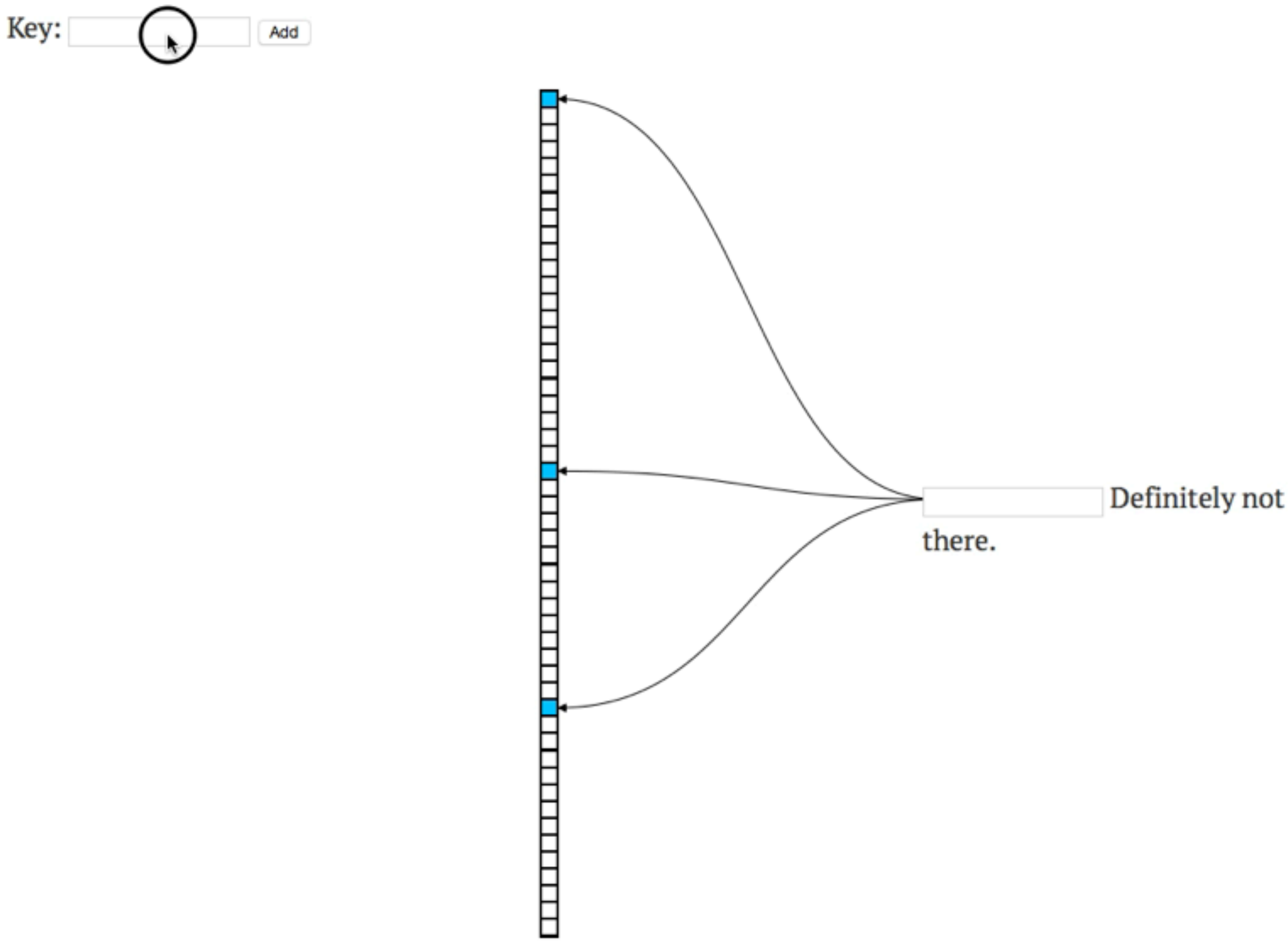
- **Given**
  - A set of hash functions $\{h_1, h_2, \ldots, h_k\}, h_i : W \to [1,n]$
  - A bit vector of size n (initialized to **0**)
- To **add** an element to W:
  - Compute $h_1(e), h_2(e), \ldots, h_k(e)$
  - Set the corresponding bits in the bit vector to 1
- To **test** whether an element is in W:
  - Compute $h_1(e), h_2(e), \ldots, h_k(e)$
  - Sum up the returned bits
  - Return TRUE if sum=k, FALSE otherwise

Usually done once in bulk with few updates.

Operation on the data stream.

http://www.jasondavies.com/bloomfilter/

Bloom filter: a demo



Bloom filter: element testing

- **Case 1**: the element is in W
  - $h_1(e), h_2(e), \ldots, h_k(e)$ are all set to 1
  - `TRUE` is returned with probability 1
- **Case 2**: the element is not in W
  - `TRUE` is returned if due to some other element all hash values are set

What is the probability of a false positive?

→ What is the probability of k bits being set to 1?

→ What is the probability of the j^{th} bit being set to 1?

Bloom filter: element testing

- **Case 1**: the element is in W
  - $h_1(e), h_2(e), \ldots, h_k(e)$ are all set to 1
  - `TRUE` is returned with probability 1

- **Case 2**: the element is not in W
  - `TRUE` is returned if due to some other element all hash values are set

$$
\begin{aligned}
P(BV_j \text{ set after } m \text{ inserts}) &= 1 - P(BV_j \text{ not set after } m \text{ inserts}) \\
&= 1 - P\left(BV_j \text{ not set after } k \times m \text{ hashes}\right) \\
&= 1 - \left(1 - \frac{1}{n}\right)^{k \times m}
\end{aligned}
$$

Bloom filter: element testing

- **Case 1**: the element is in W
  - $h_1(e), h_2(e), \ldots, h_k(e)$ are all set to 1
  - `TRUE` is returned with probability 1

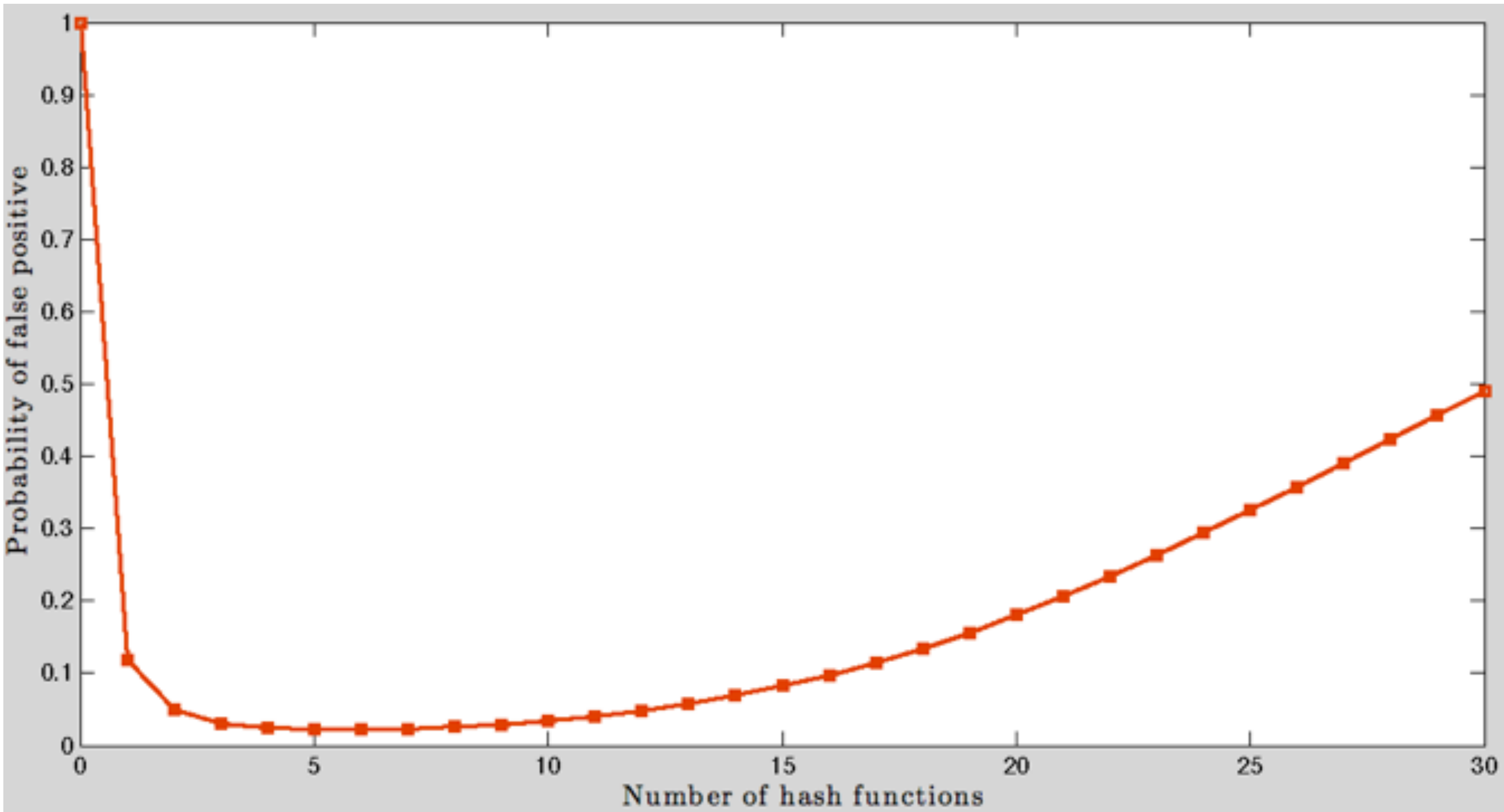
- **Case 2**: the element is not in W
  - `TRUE` is returned if due to some other element all hash values are set

$$
\begin{aligned}
P(BV_j \text{ set after } m \text{ inserts}) &= 1 - P(BV_j \text{ not set after } m \text{ inserts}) \\
&= 1 - P\left(BV_j \text{ not set after } k \times m \text{ hashes}\right) \\
&= 1 - \left(1 - \frac{1}{n}\right)^{k \times m}
\end{aligned}
$$

$$
P(\textit{false positive}) = \left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^{k}
$$

Bloom filter: how many hash functions are useful?

Example: $m = 10^9$ whitelisted IP addresses and $n = 8 \times 10^9$ bits in memory



Bloom filter tricks

- Union of two Bloom filters of the same type in terms of hash functions and bits

OR the two bit vectors.

- To half the size of a Bloom filter with a filter size the power of 2

OR first and second half together.
When hashing the higher order bit can be masked.

- Bloom filter deletions?

Not possible in the standard setup.
Solution: counting bloom filters (instead of bits use counters that increment/decrement).