

DATA STREAMS

Data streaming

- 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 stream models

- Massively long input stream
- Basic model: $\sigma = \langle a_1, a_2, a_3, \dots, a_m \rangle$
with elements drawn from $[n] := 1, 2, \dots, n$
- Space complexity goal: s bits of random-access memory with

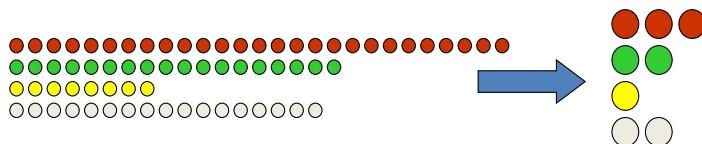
$$s = o(\min\{m, n\})$$

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

“holy grail”

$$s = \text{poly log}(\min(m, n))$$

“reality”



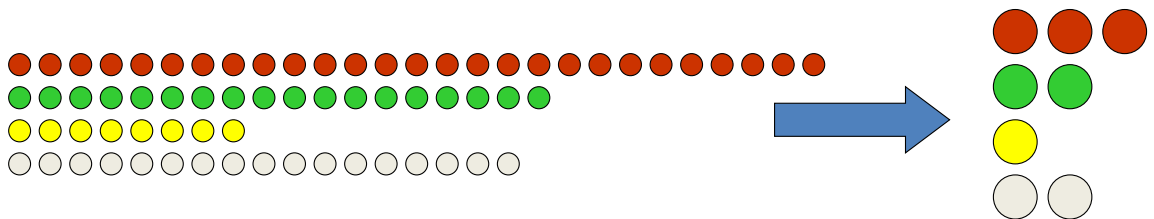
SAMPLING

Sampling

- 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

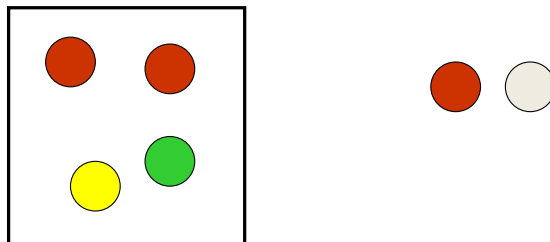
Sampling from a Data Stream

- Fundamental problem: sample m items uniformly from stream
 - Useful: approximate costly computation on small sample
- Challenge: don't know how long stream is
 - So when/how often to sample?
- Two solutions, apply to different situations:
 - Reservoir sampling (dates from 1980s?)
 - Min-wise sampling (dates from 1990s?)



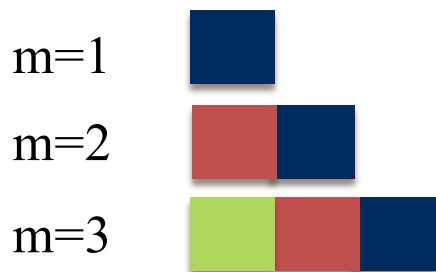
Reservoir Sampling

- Sample first m items
- Choose to sample the i 'th item ($i > m$) with probability $p_i = m/i$
- If sampled, randomly replace a previously sampled item
- Optimization: when i gets large, compute which item will be sampled next, skip over intervening items



Reservoir sampling, example

- Toy example with $k=1$



$$\begin{aligned} P(\blacksquare) &= 1 \times \frac{1}{2} \times \frac{2}{3} = \frac{1}{3} \\ P(\blacksquare) &= \frac{1}{2} \times \frac{2}{3} = \frac{1}{3} \\ P(\blacksquare) &= \frac{1}{3} \end{aligned}$$

Reservoir Sampling - Analysis

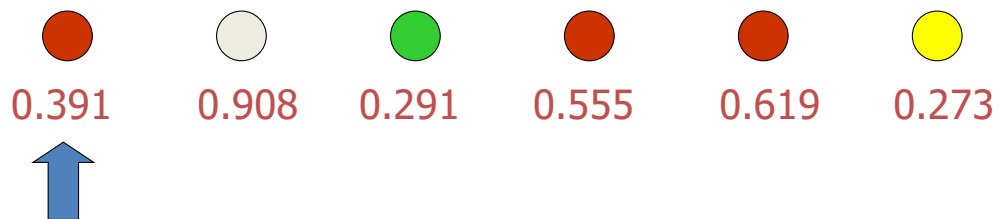
- Analyze simple case: sample size $m = 1$
- Probability i 'th item is the sample from stream length n :
 - Prob. i is sampled on arrival \times prob. i survives to end

$$\frac{1}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \cdots \frac{n-2}{n-1} \times \frac{n-1}{n} = 1/n$$

Case for $m > 1$ is similar, easy to show uniform probability

Reservoir Sampling via Order Sampling

- a.k.a. bottom-k sample, min-hashing, min-wise sampling, ...
- Uniform sampling of stream into reservoir of size k
- Each arrival n: generate one-time random value $r_n \in U[0,1]$
 - r_n also known as hash, rank, tag...
- Store k items with the smallest random tags



Each item has same chance of least tag, so uniform

Fast to implement via priority queue

Can run on multiple input streams separately, then merge

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.”

Finding Frequent Items in Data Streams by Graham Cormode & Marios Hadjieleftheriou.

Examples

Packets on the Internet

Frequent items:

most popular destinations

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)?








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

In this stream, the last item is kept.

A **second pass** is needed to verify if the stored item is indeed the absolute majority item (count every occurrence of 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 $> \frac{m}{k}$)

```
 $c[1, \dots, (k-1)] = 0; T \leftarrow \emptyset;$ 
```

```
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)

$k = 3$

$c = 0$

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



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

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)

$k = 3$

$c = 0$



Green is estimated
to have occurred
once.





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

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

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)

space complexity

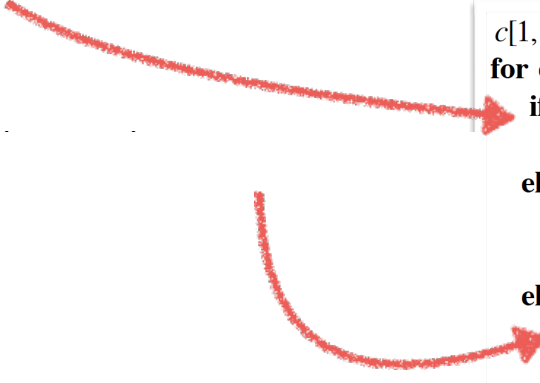
- Implementation: associative array using a balanced binary search tree
- Each key has a max. value of n , each counter has a max. value of m
- At most $(k-1)$ key/counter pairs in memory at any time

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

FREQUENT algorithm (Misra-Gries)

answer quality of frequency estimates

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



```
c[1,..(k-1)] = 0; T ← ∅;  
for each i :  
  if i ∈ T :  
    ci ← ci + 1;  
  else if |T| < k - 1 :  
    T ← T ∪ {i};  
    ci ← 1;  
  else for all j ∈ T :  
    cj ← cj - 1;  
    if cj = 0 :  
      T ← T \ {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

```
 $c[1, \dots, (k-1)] = 0; T \leftarrow \emptyset;$ 
```

```
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:
```

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

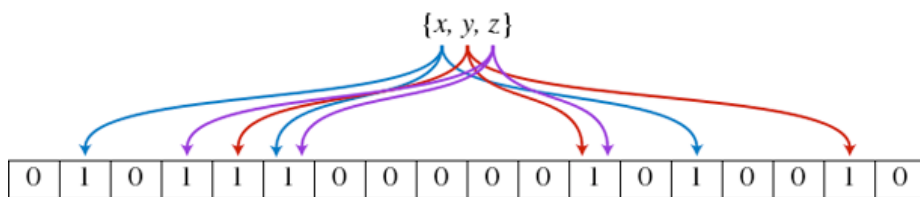
```
     $c_i \leftarrow c_j + 1;$ 
```

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

FREQUENT algorithm (Space Saving)

Analysis

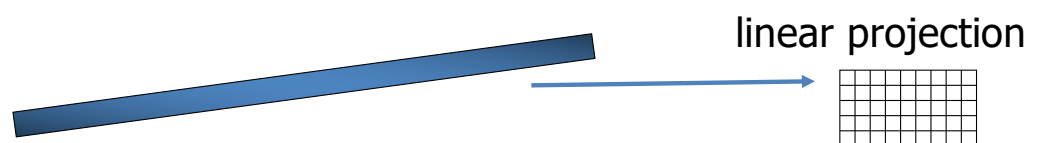
- Smallest counter value min is at most m/k
 - Counters sum to m by induction
 - k counters, so average is m/k : smallest cannot be bigger
- True count of an uncounted item is between 0 and min
 - Proof by induction, true initially, min cannot decrease
 - All counts are overestimates: count of any item is off by at most m/k
- Any item x whose true count $> m/k$ is stored
 - By contradiction: last time x was removed, with count $\leq \min_t$
 - x does not occur afterwards, so:
 - Count of $x > m/k \geq \min \geq \min_t$, and would not be removed



HASHING


Sketches

- Not every problem can be solved with sampling
 - **Example**: counting how many distinct items in the stream
 - If a large fraction of items aren't sampled, don't know if they are all same or all different
- Other techniques take advantage that the algorithm can “see” all the data even if it can't “remember” it all
- **“Sketch”**: essentially, a linear transform of the input
 - Model stream as defining a vector, sketch is result of multiplying stream vector by an (implicit) matrix



Bloom filter

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



- **Given**

- A set of hash functions $\{h_1, h_2, \dots, h_k\}, h_i : W \rightarrow [1, n]$
- A bit vector of size n (initialized to **0**)

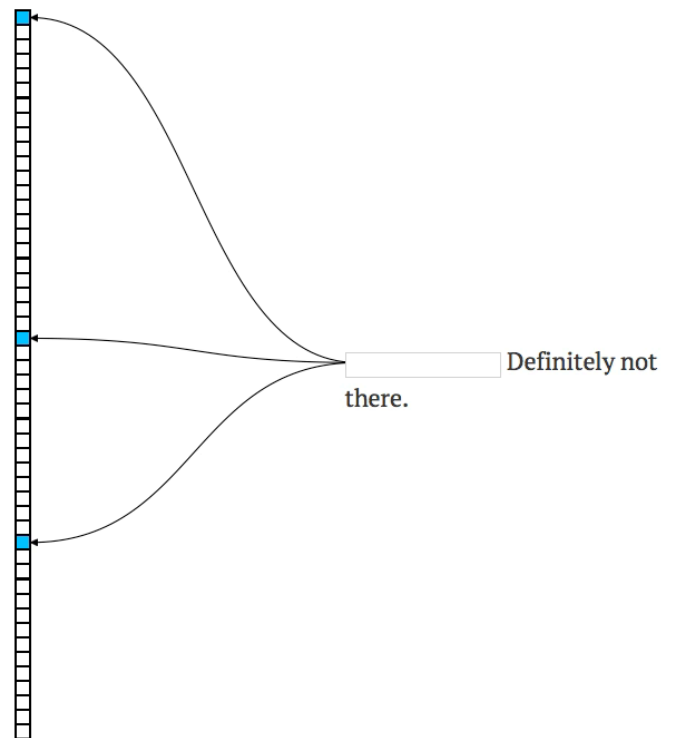
Usually done once in bulk with few updates.

Operation on the data stream.

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

Bloom filter: a demo

Key:

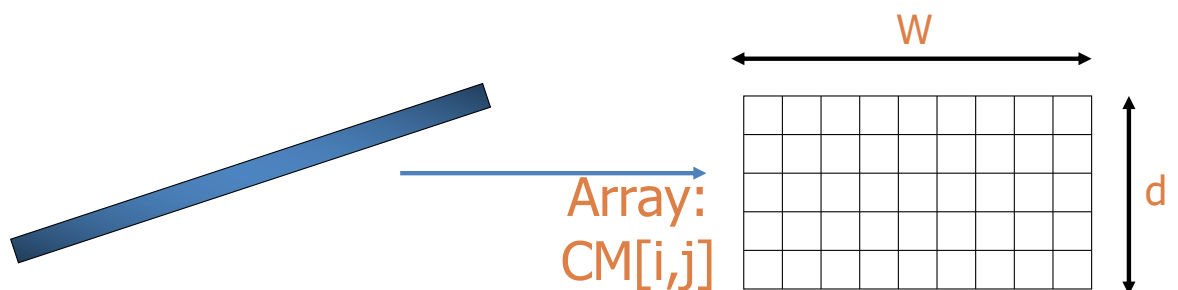


Bloom filter: element testing

- **Case 1:** the element is in W
 - $h_1(e), h_2(e), \dots, h_k(e)$ 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

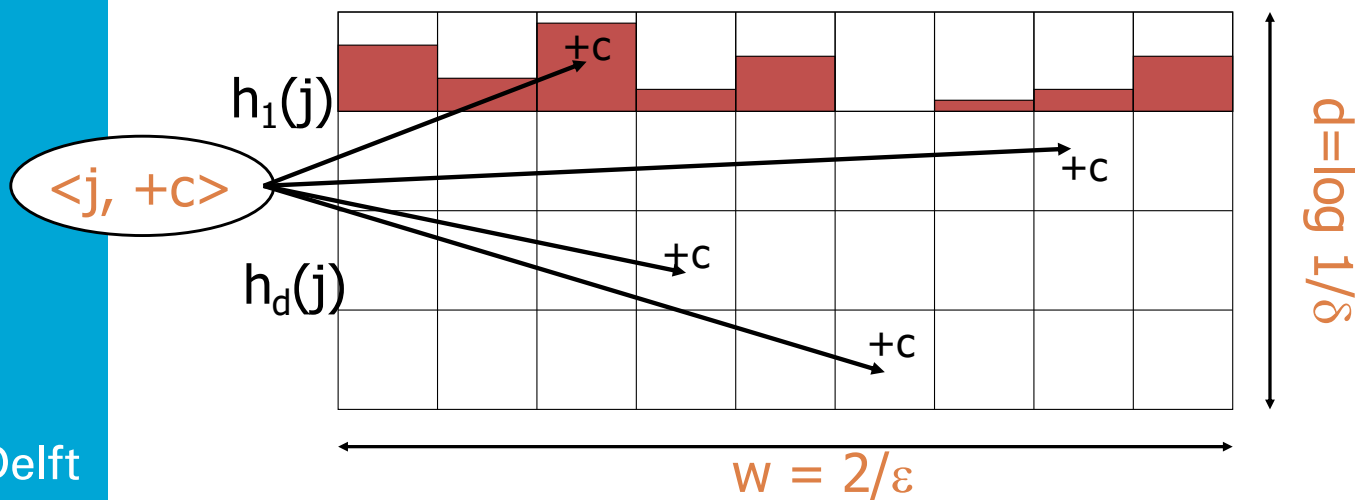
Count-Min Sketch

- Simple sketch idea, can be used for as the basis of many different stream mining tasks
 - Join aggregates, range queries, moments, ...
- Creates a small summary as an array of $w \times d$ in size
- Use d hash functions to map vector entries to $[1..w]$



CM Sketch Structure

- Assume a stream of length m
- Modeled using a vector $A[]$ of counts
- Entries from $A[]$ are mapped to one bucket per row
- Counts merged in buckets are summed up
- Estimate $A[j]$ by taking $\min_k \{ CM[k, h_k(j)] \}$



CM Sketch Guarantees

- CM sketch guarantees approximation error on point queries less than ϵm in space $O(1/\epsilon \log 1/\delta)$
 - Probability of more error is less than $1-\delta$
- How?
 - Counts are *biased (overestimates)* due to collisions
 - **Use independence across rows** to boost the confidence for the $\min\{\}$ estimate

CM Sketch Analysis

Estimate $A'[j] = \min_k \{ CM[k, h_k(j)] \}$

- Analysis: In k 'th row, $CM[k, h_k(j)] = A[j] + X_{k,j}$
 - $X_{k,j} = \sum A[i \neq j] \mid h_k(i) = h_k(j)$
 - $E[X_{k,j}] = \sum A[i \neq j] \cdot \Pr[h_k(i) = h_k(j)]$ (recall $2/\epsilon$ columns)
 $\leq (\epsilon/2) \cdot \sum A[i \neq j] \leq \epsilon m/2$ (pairwise independence)
 - $\Pr[X_{k,j} \geq \epsilon m] = \Pr[X_{k,j} \geq 2E[X_{k,j}]] \leq 1/2$ by **Markov inequality**
 $\Pr(X \geq a) \leq E(x) / a$
- $\Pr[A'[j] \geq A[j] + \epsilon m] = \Pr[\forall k. X_{k,j} > \epsilon m] \leq 1/2^{\log 1/\delta} = \delta$
- Final result: with certainty $A[j] \leq A'[j]$ and
with probability at least $1-\delta$: $A'[j] < A[j] + \epsilon m$

FM-sketch (Flajolet-Martin)

- Approach: hash data stream elements uniformly to N bit values, i.e.:
- Task: Given a data stream of, estimate the **number of distinct elements** occurring in it.
- Assumption: the larger the number of distinct elements in the stream, the more distinct the occurring hash values, and the more likely one with an **unusual property** appears

$$h : a_i \rightarrow \{0, 1\}^N$$

FM-sketch

- One possibility of interpreting unusual is the hash tail: the number of 0's a binary hash value ends in

100110101110

100110101100

100110000000

for all $a_i \in S$ (our stream):

$$h(a_i) \rightarrow \{0, 1\}^N$$

maximum hash tail seen so far

$$K = \max_{a_i \in S} \text{tail}(h(a_i))$$

$$\text{return } |\hat{S}| = 2^K$$

N must be **long enough**;
there must be **more possible results** of the
hash function **than elements** in the
universal set.

FM-sketch (Flajolet-Martin)

Int

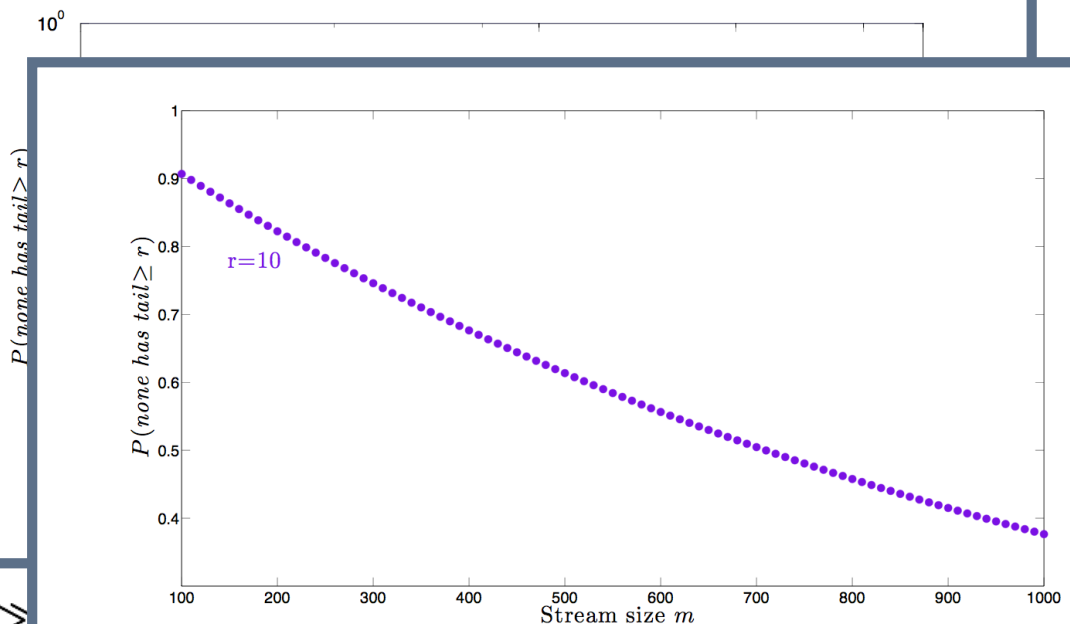
F

Wh

F

if $m \gg$

if $m \ll 2^r$: the prob. of finding a tail $\geq r$ reaches 0



FM-sketch (Flajolet-Martin)

Int

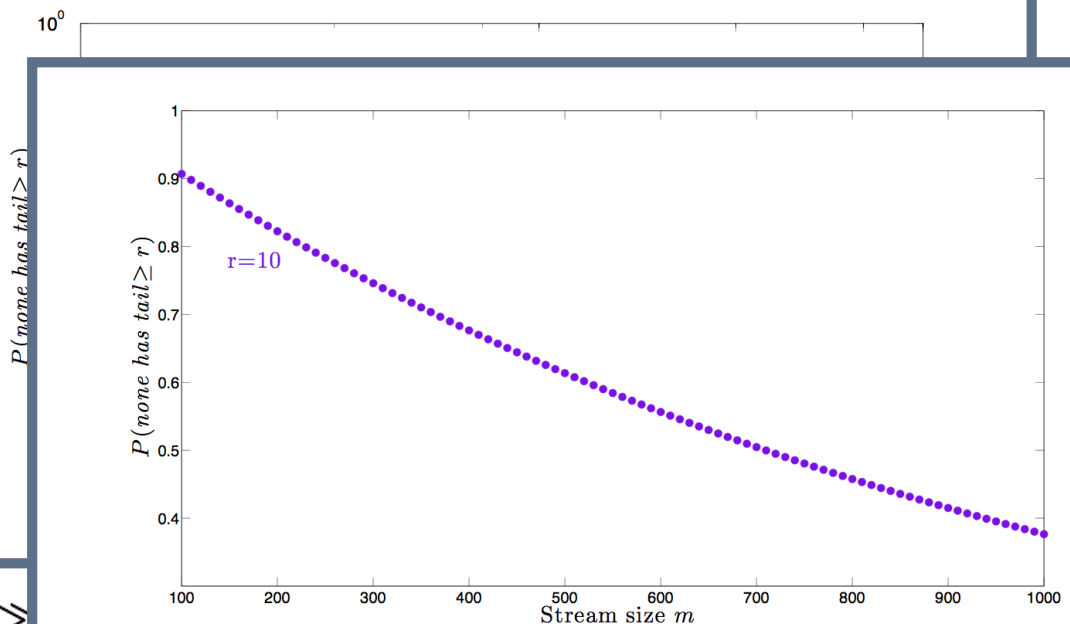
F

Wh

F

if $m \gg$

if $m \ll 2^r$: the prob. of finding a tail $\geq r$ reaches 0



LOCALITY SENSITIVE HASHING AND SUFFICIENT STATISTICS

Hashing, recap

- Approximate queries (questions on counts) over large data streams using very limited memory
- Sampling:
 - Selectively storing elements, keeping certain properties intact (e.g. a uniform distribution, ...)
- Hashing:
 - Store multiple (counts of) hashes of incoming elements
 - Answer queries using stored hashes
 - Hashes may overlap, causing errors in query answers

Hash using Signatures

- Key idea: “hash” each column C to a small signature $\text{Sig}(C)$, such that:
 1. $\text{Sig}(C)$ is small enough that we can fit a signature in main memory for each column.
 2. $\text{Sim}(C1, C2)$ is the same as “similarity” of $\text{Sig}(C1)$ and $\text{Sig}(C2)$.
 - i.e., *$\text{Sig}()$ is locality sensitive!*

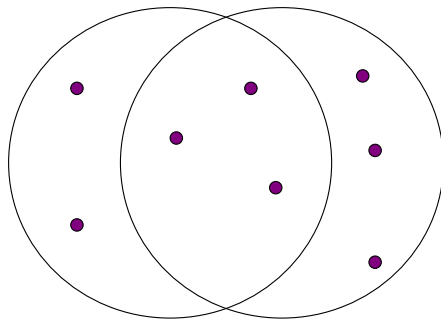
Hash using Signatures

- Key idea: “hash” each column C to a small signature $\text{Sig}(C)$, such that:
 1. $\text{Sig}(C)$ is small enough that we can fit a signature in main memory for each column.
 2. $\text{Sim}(C1, C2)$ is the same as “similarity” of $\text{Sig}(C1)$ and $\text{Sig}(C2)$.
 - i.e., *$\text{Sig}()$ is locality sensitive!*

Why would this be useful?

Jaccard Similarity of Sets

- The Jaccard similarity of two sets is the size of their intersection divided by the size of their union.
 - $\text{Sim}(C1, C2) = |C1 \cap C2| / |C1 \cup C2|$.



3 in intersection.
8 in union.
Jaccard similarity
= 3/8

Four Types of Rows

- Given columns C1 and C2, rows may be classified as:

	C1	C2
a	1	1
b	1	0
c	0	1
d	0	0

- Also, $a = \# \text{ rows of type a}$, etc.
- Note $\text{Sim}(C1, C2) = a / (a + b + c)$.

Minhashing

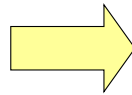
- Imagine the rows permuted randomly.
- Define “hash” function $h(C)$
 - the row number of the first (in the permuted order) row in which column C has 1.
- Use several (e.g., 100) independent hash functions to create a signature.

Minhashing Example

Input matrix

1	4	3
3	2	4
7	1	7
6	3	6
2	6	1
5	7	2
4	5	5

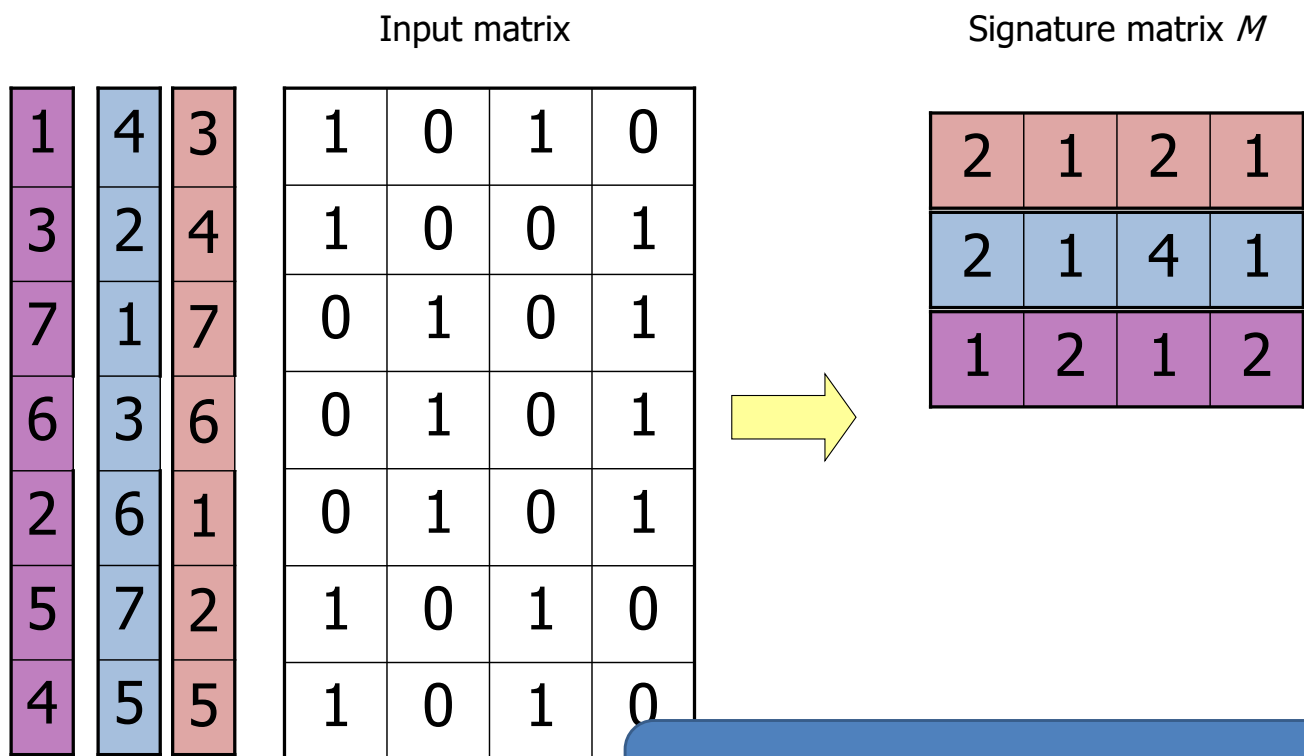
1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0



Signature matrix M

2	1	2	1
2	1	4	1
1	2	1	2

Minhashing Example



Q: Why would this be useful?

Surprising Property

- The probability (over all permutations of the rows) that $h(C1) = h(C2)$ is the same as $\text{Sim}(C1, C2)$.
- Both are $a / (a + b + c)!$
 - (a = number of a-type rows, etc.)
- **Why?**

Surprising Property

- The probability (over all permutations of the rows) that $h(C1) = h(C2)$ is the same as $\text{Sim}(C1, C2)$.
- Both are $a / (a + b + c)!$
 - (a = number of a-type rows, etc.)
- Why?
 - Look down the permuted columns $C1$ and $C2$ until we see a 1.
 - Every row has equal probability to be the first with a 1.
 - If it is a type-a row, then $h(C1) = h(C2)$.
 - If it is a type-b or type-c row, then not.

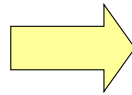
Min Hashing – Example

Input matrix

1	4	3	1	0	1	0
3	2	4	1	0	0	1
7	1	7	0	1	0	1
6	3	6	0	1	0	1
2	6	1	0	1	0	1
5	7	2	1	0	1	0
4	5	5	1	0	1	0

Signature matrix M

2	1	2	1
2	1	4	1
1	2	1	2



Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

Implementation: Hashing!

- Suppose 1 billion rows.
 - Hard to pick a random permutation from 1...billion.
- A good approximation to permuting rows:
 - pick 100 (?) hash functions.
- For each column c and each hash function h_i , keep a “slot” $M(i, c)$.
- Intent: $M(i, c)$ will become the smallest value of $h_i(r)$ for which column c has 1 in row r .

Example

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

$$h(x) = x \bmod 5$$
$$g(x) = 2x+1 \bmod 5$$

	Sig1	Sig2
$h(1) = 1$	1	-
$g(1) = 3$	3	-
$h(2) = 2$	1	2
$g(2) = 0$	3	0
$h(3) = 3$	1	2
$g(3) = 2$	2	0
$h(4) = 4$	1	2
$g(4) = 4$	2	0
$h(5) = 0$	1	0
$g(5) = 1$	2	0

Locality Sensitive Hashing

- The basic idea behind LSH is to project the data into a **low-dimensional binary space**; that is, each data point is mapped to a b-bit vector, the *hash key*
- Each hash function h must satisfy the
 - **sensitive hashing property:**

$$\Pr[h(\mathbf{x}_i) = h(\mathbf{x}_j)] = \text{sim}(\mathbf{x}_i, \mathbf{x}_j)$$

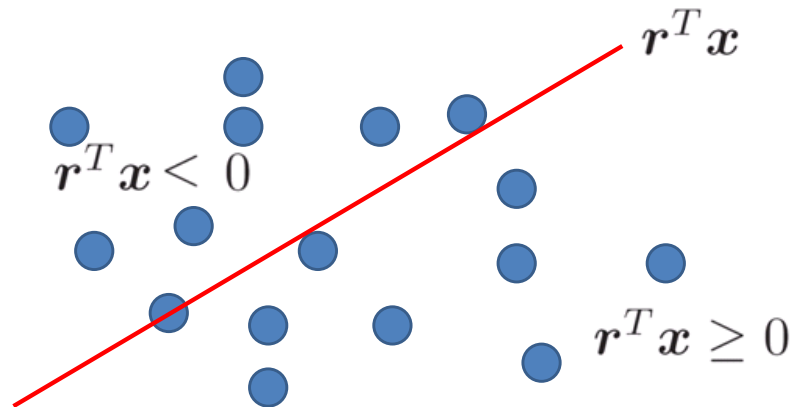
- Where $\text{sim}(\mathbf{x}_i, \mathbf{x}_j) \in [0, 1]$ is the similarity function of interest

LSH functions for dot products

The hashing function of LSH to produce Hash Code

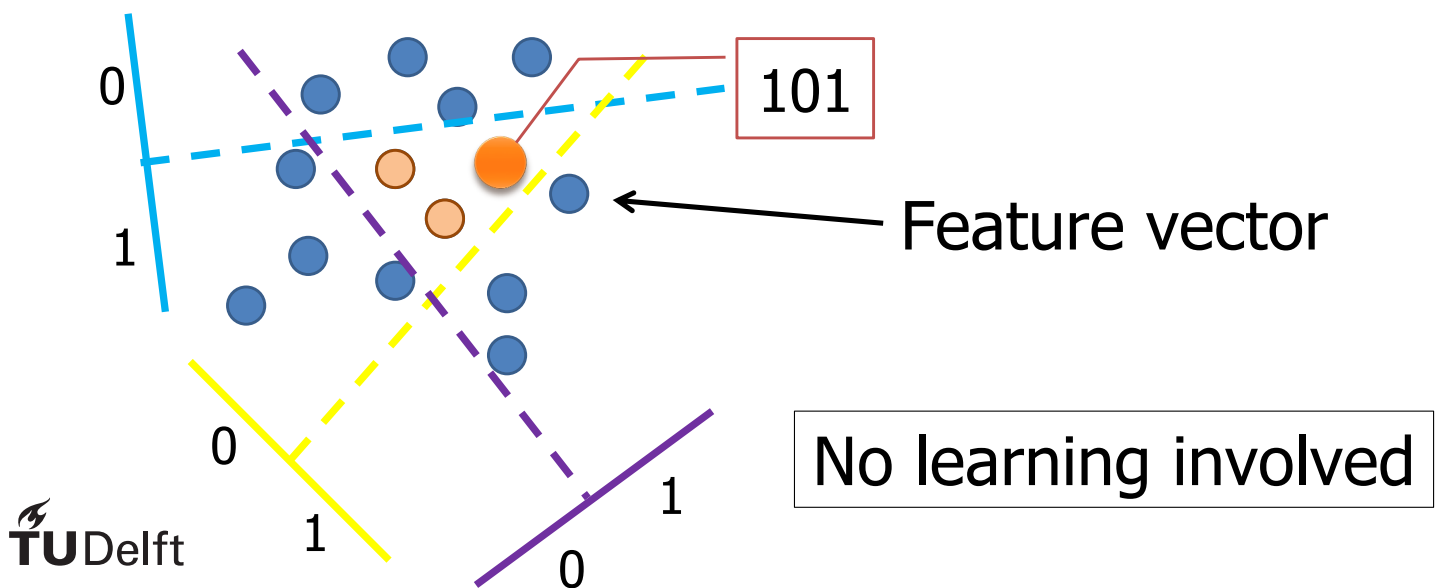
$$h_r(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{r}^T \mathbf{x} \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

$\mathbf{r}^T \mathbf{x} \geq 0$ is a hyperplane separating the space

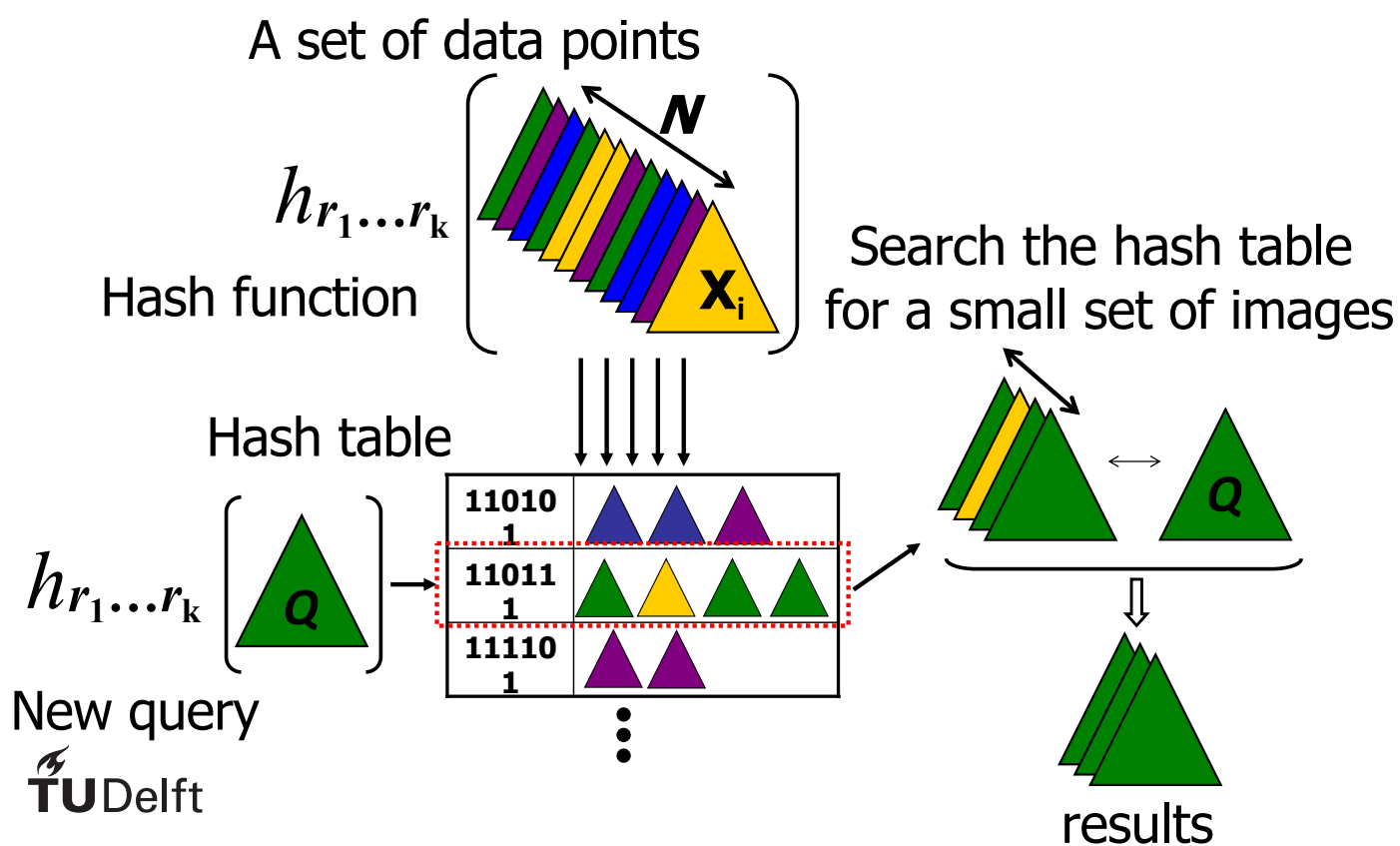


Locality Sensitive Hashing

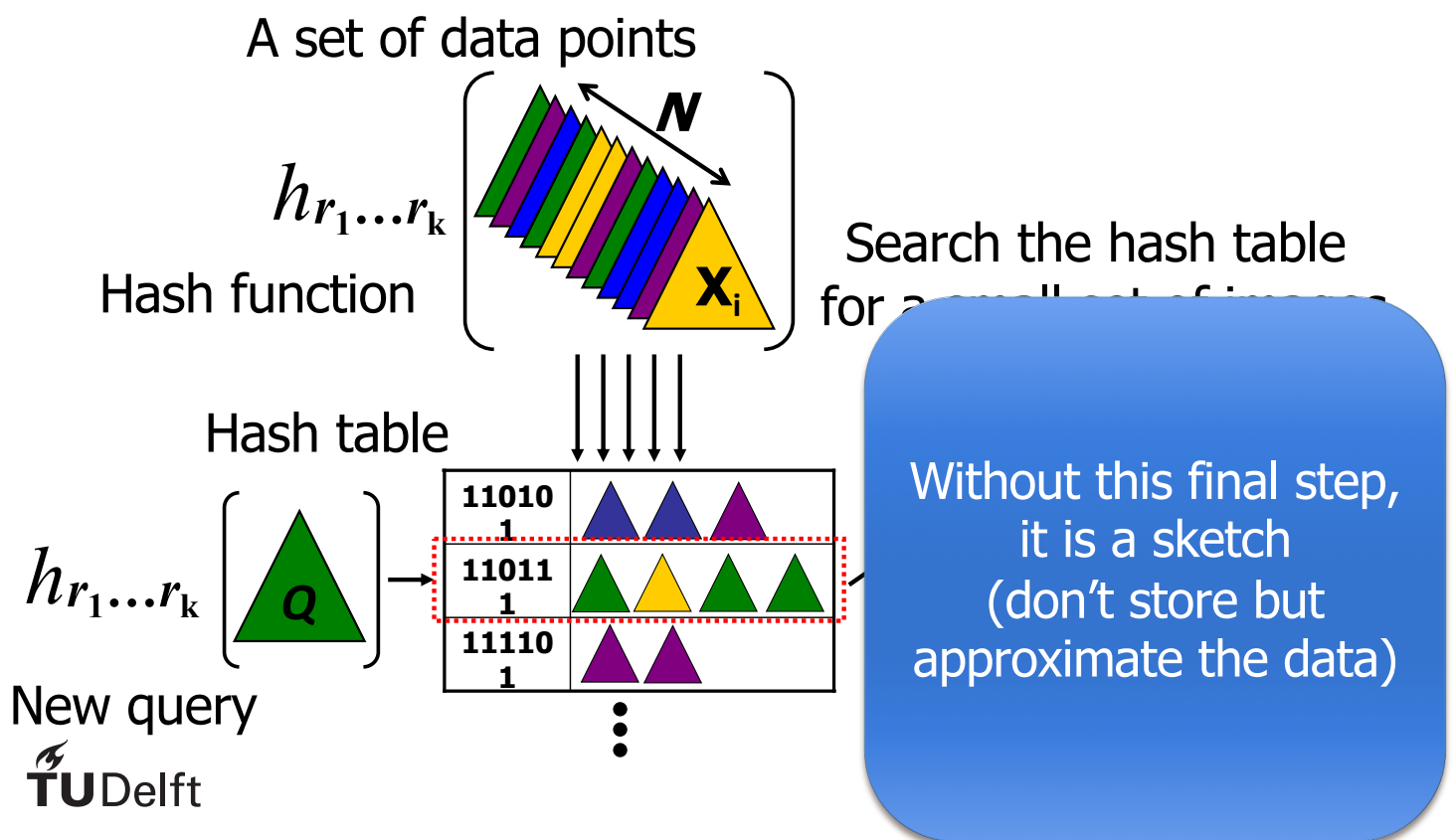
- Take random projections of data
- Quantize each projection with few bits



How to search from hash table?



How to search from hash table?



Sketching, Hashing, Sampling Summary

- Sampling, hashing, and sketching ideas are at the heart of many stream mining algorithms
 - Moments/join aggregates, histograms, wavelets, top-k, frequent items, other mining problems, ...
- A sample is a quite **general representative** of the data set; sketches tend to be *specific to a particular purpose*
 - FM sketch for count distinct, CM/AMS sketch for joins / moment estimation, LSH for distances, ...
- Sample implementations available on the web
 - <http://www.cs.rutgers.edu/~muthu/massdal-code-index.html>
 - <http://www.mit.edu/~andoni/LSH>
- ***All provide approximations of true counts!***

Sketching, Hashing, Sampling Summary

- Sampling, hashing, and sketching ideas are at the heart of many stream mining algorithms
 - Moments/join aggregates, histograms, wavelets, top-k, frequent items, other mining problems, ...
- A sample is a quite **general representative** of the data set; sketches tend to be *specific to a particular purpose*
 - FM sketch for count distinct, CM/AMS sketch for joins / moment estimation, LSH for distances, ...
- Sample implementations available on the web
 - <http://www.cs.rutgers.edu/~muthu/massdal-code-index.html>
 - <http://www.mit.edu/~an>
- *All provide approxima*

Hashing is COOL!