COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 6.2: Mining Data Streams II

Part 3: Filtering Data Streams

Filtering Data Streams

- Each element of data stream is a tuple
- Given a list of keys **S**
- **Determine which tuples of stream are in *S***

- Obvious solution: Hash table
  - But suppose we **do not have enough memory** to store all of ***S*** in a hash table
    - E.g., we might be processing millions of filters on the same stream

Applications

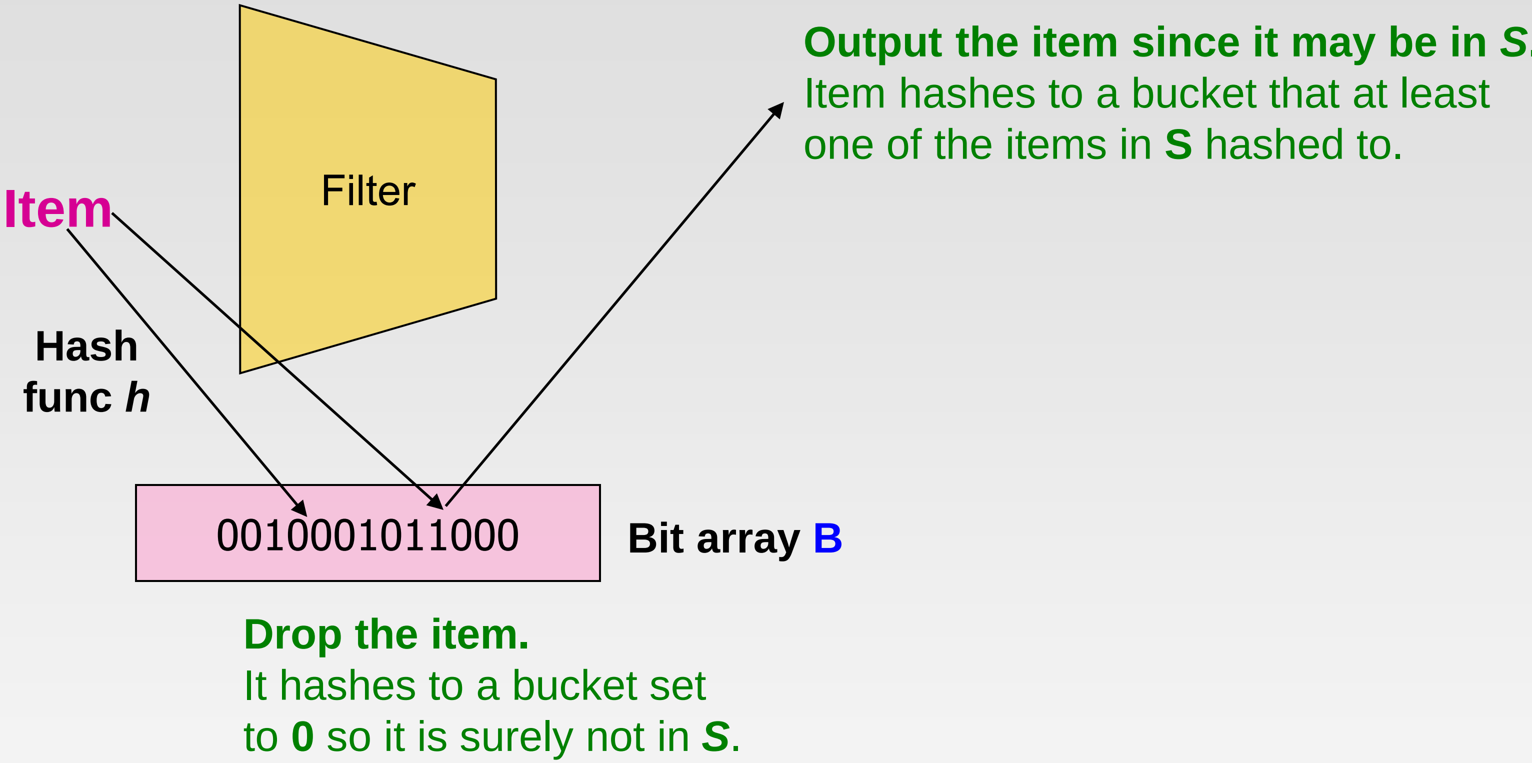
- Example: Email spam filtering
  - We know 1 billion “good” email addresses
  - If an email comes from one of these, it is **NOT** spam

- Publish-subscribe systems
  - You are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches user’s interest

First Cut Solution (1)

- Given a set of keys ***S*** that we want to filter
- Create a **bit array *B*** of ***n*** bits, initially all ***0*s**
- Choose a **hash function *h*** with range **[*0,n*)**
- Hash each member of $s \in S$ to one of ***n*** buckets, and set that bit to **1**, i.e., ***B[h(s)]=1***
- Hash each element ***a*** of the stream and output only those that hash to bit that was set to **1**
  - **Output *a* if B[h(a)] == 1**

First Cut Solution (2)



- **Creates false positives but no false negatives**
  - If the item is in ***S*** we surely output it, if not we may still output it

First Cut Solution (3)

- **|S| = 1 billion email addresses**
  |B|= 1GB = 8 billion bits

- If the email address is in ***S***, then it surely hashes to a bucket that has the big set to **1**, so it always gets through (***no false negatives***)
  - False negative: a result indicates that a condition failed, while it actually was successful

- Approximately 1/8 of the bits are set to 1, so about 1/8th of the addresses not in S get through to the output (***false positives***)
  - False positive: a result that indicates a given condition has been fulfilled, when it actually has not been fulfilled
  - Actually, less than 1/8th, because more than one address might hash to the same bit
  - Since the majority of emails are spam, eliminating 7/8th of the spam is a significant benefit

Analysis: Throwing Darts (1)

- More accurate analysis for the number of **false positives**
- **Consider:** If we throw ***m*** darts into ***n*** equally likely targets, **what is the probability that a target gets at least one dart?**
- **In our case:**
  - **Targets** = bits/buckets
  - **Darts** = hash values of items

Analysis: Throwing Darts (2)

- We have ***m*** darts, ***n*** targets
- **What is the probability that a target gets at least one dart?**

Equals **1/e** as **n** $\rightarrow\infty$

Equivalent

$$1 - \boxed{\underbrace{\left(1 - 1/n\right)}_{\text{Probability some target X not hit by a dart}}}^{\,n(\,m\,/\,n)}$$

$$1 - e^{-m/n}$$

Probability some target **X** not hit by a dart

Probability at least one dart hits target **X**

Analysis: Throwing Darts (3)

- **Fraction of 1s in the array B**

  = probability of false positive = $1 - e^{-m/n}$

- **Example:** $\mathbf{10^9}$ darts, $\mathbf{8 \cdot 10^9}$ targets
  - Fraction of **1s** in **B = $1 - e^{-1/8}$ = 0.1175**
    - Compare with our earlier estimate: **1/8 = 0.125**

Bloom Filter

- Consider: **|S| = *m*, |B| = *n***
- Use ***k*** independent hash functions $h_1, \dots, h_k$
- **Initialization:**
  - Set **B** to all **0s**
  - Hash each element $s \in S$ using each hash function h_i, set **B[$h_i(s)$] = 1** (for each $i = 1,.., k$)
- **Run-time:**
  - When a stream element with key ***x*** arrives
    - If **B[$h_i(x)$] = 1** <u>**for all**</u> **i = 1,..., k** then declare that ***x*** is in ***S***
      - That is, ***x*** hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element ***x***

Bloom Filter

Start with an n bit array, filled with 0s.

B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S for k times. If $H_i(x_j) = a$, set $B[a] = 1$.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

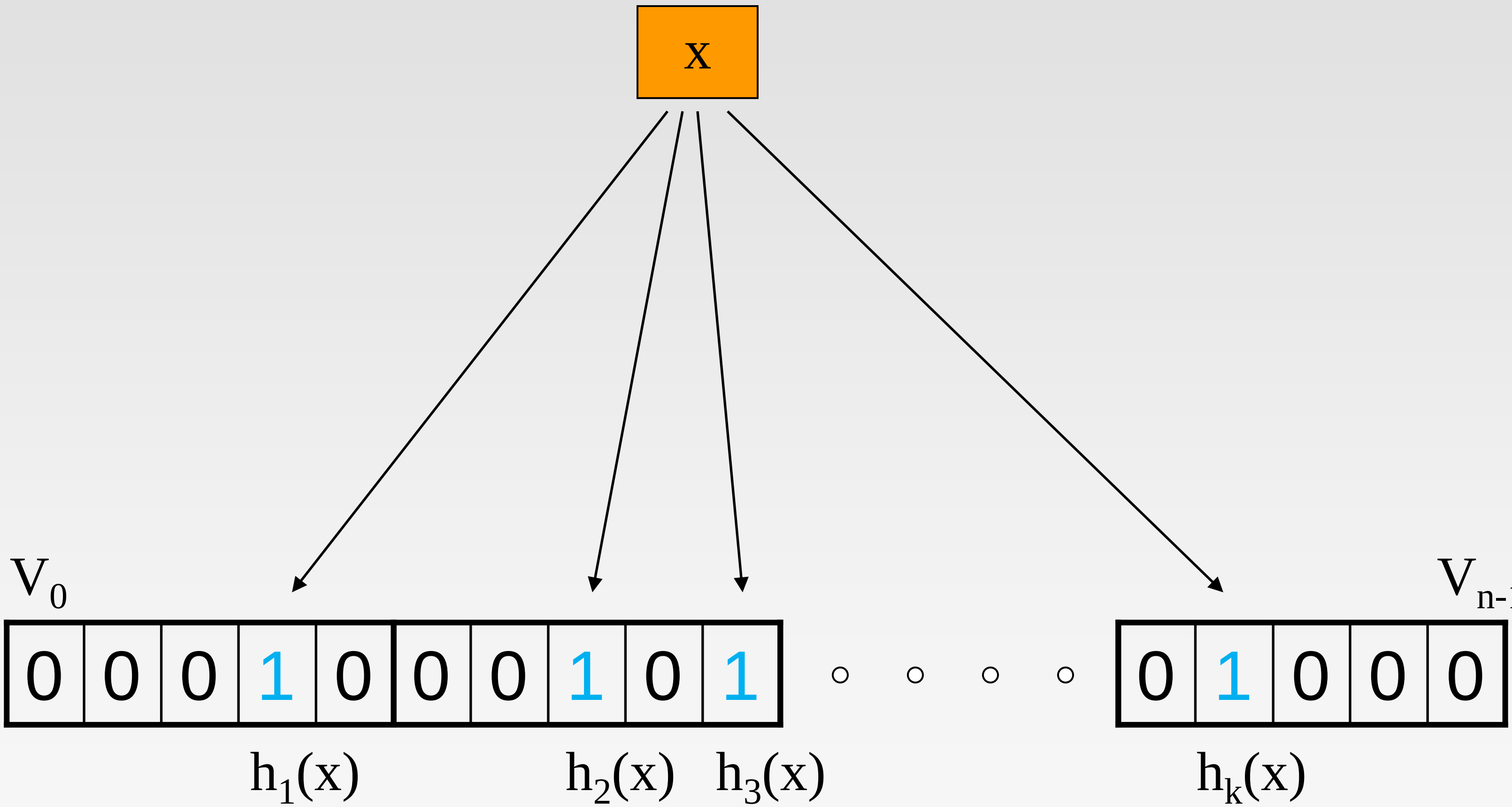
To check if y is in S, check B at $H_i(y)$. All k values must be 1.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

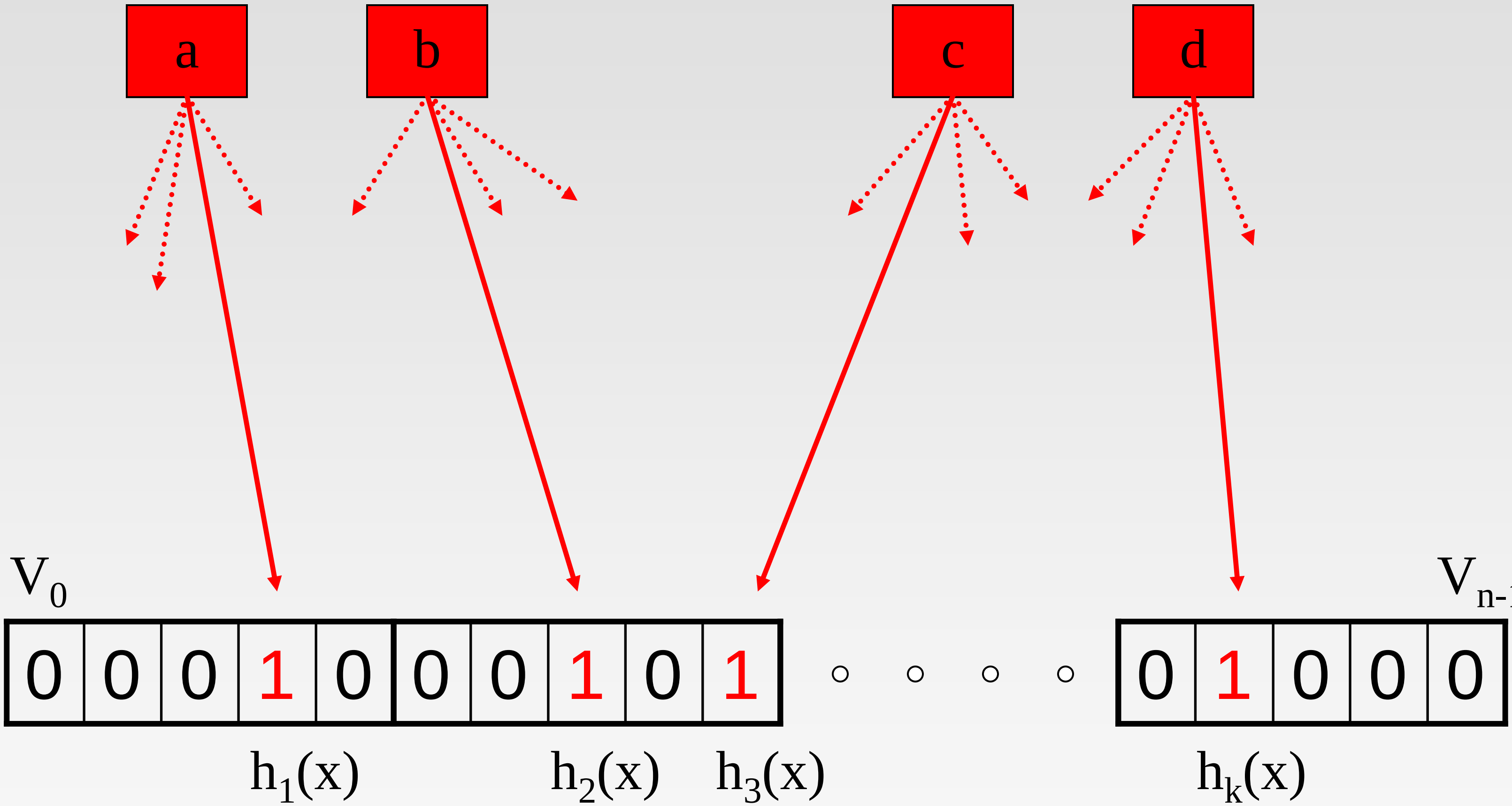
Possible to have a false positive; all k values are 1, but y is not in S.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Bloom Filter Hashing



Bloom Errors



x didn’t appear, yet its bits are already set

Bloom Filter Example

- Consider a Bloom filter of size m=10 and number of hash functions k=3. Let H(x) denote the result of the three hash functions.
- The 10-bit array is initialized as below

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

- Insert x_0 with $H(x_0) = \{1, 4, 9\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

- Insert x_1 with $H(x_1) = \{4, 5, 8\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

- Query y_0 with $H(y_0) = \{0, 4, 8\}$ => ???
- Query y_1 with $H(y_1) = \{1, 5, 8\}$ => ??? **False positive!**

- Another Example: https://llimllib.github.io/bloomfilter-tutorial/

Bloom Filter – Analysis

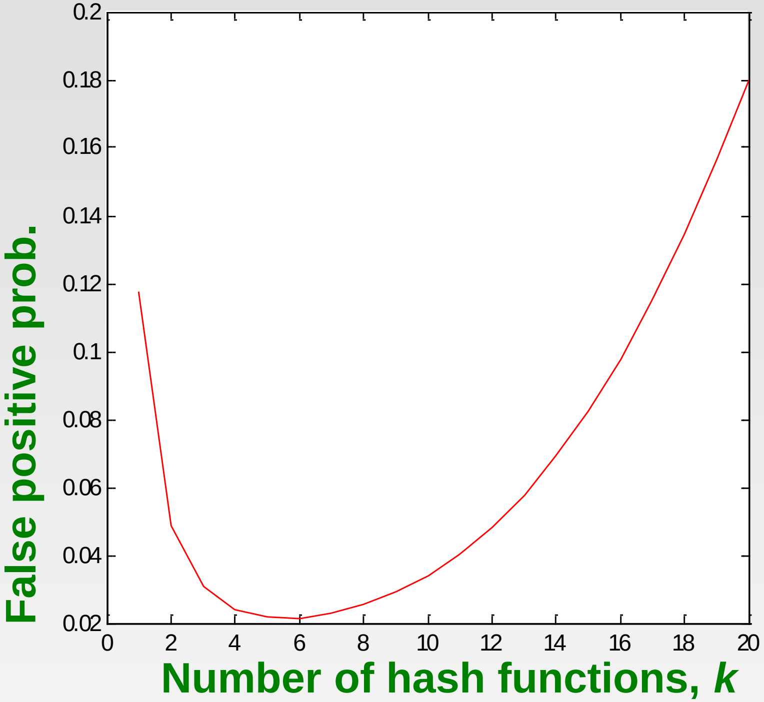
- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of **1**s is $(1 - e^{-km/n})$

- But we have k independent hash functions and we only let the element x through **if all** k hash element x to a bucket of value **1**

- So, false **positive probability = $(1 - e^{-km/n})^k$**

Bloom Filter – Analysis (2)

- *m* = 1 billion, *n* = 8 billion
  - **k = 1**: $(1 - e^{-1/8})$ = **0.1175**
  - **k = 2**: $(1 - e^{-1/4})^2$ = **0.0493**



- **What happens as we keep increasing *k*?**

- "Optimal" value of ***k***: ***n/m* ln(2)**
  - **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**
    - **Error at k = 6**: $(1 - e^{-6/8})^6$ = **0.02158**

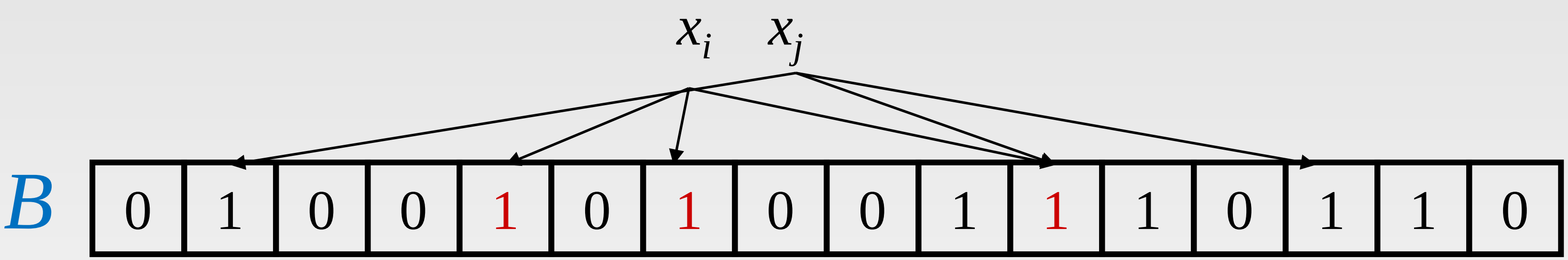
Bloom Filter: Wrap-up

- Bloom filters guarantee no false negatives, and use limited memory
  - Great for pre-processing before more expensive checks
- Suitable for hardware implementation
  - Hash function computations can be parallelized
- Is it better to have **1** big **B** or ***k*** small **B**s?
  - **It is the same:** $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping **1 big B** is simpler

Handling Deletions

- Bloom filters can handle insertions, but not deletions.

- If deleting x_i means resetting 1s to 0s, then deleting x_i will "delete" x_j.



- Can Bloom filters handle deletions?
  - Use Counting Bloom Filters to track insertions/deletions

Counting Bloom Filters

Start with an n bit array, filled with 0s.

B																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S for k times. If $H_i(x_j) = a$, add 1 to $B[a]$.

B																
	0	3	0	0	1	0	2	0	0	3	2	1	0	2	1	0

To delete x_j decrement the corresponding counters.

B																
	0	2	0	0	0	0	2	0	0	3	2	1	0	1	1	0

Can obtain a corresponding Bloom filter by reducing to 0/1.

B																
	0	1	0	0	0	0	1	0	0	1	1	1	0	1	1	0

Part 4: Counting Data Streams (FM-Sketch)

Counting Distinct Elements

- Problem:
  - Data stream consists of a universe of elements chosen from a set of size ***N***
  - Maintain a count of the number of distinct elements seen so far

- Example:

Data stream: 3 2 5 3 2 1 7 5 1 2 3 7

Number of distinct values: 5

- Obvious approach: Maintain the set of elements seen so far
  - That is, keep a hash table of all the distinct elements seen so far
  - Not practical if we only have fixed-size storage

Applications

- How many different words are found among the Web pages being crawled at a site?
  - Unusually low or high numbers could indicate artificial pages (spam?)

- How many different Web pages does each customer request in a week?

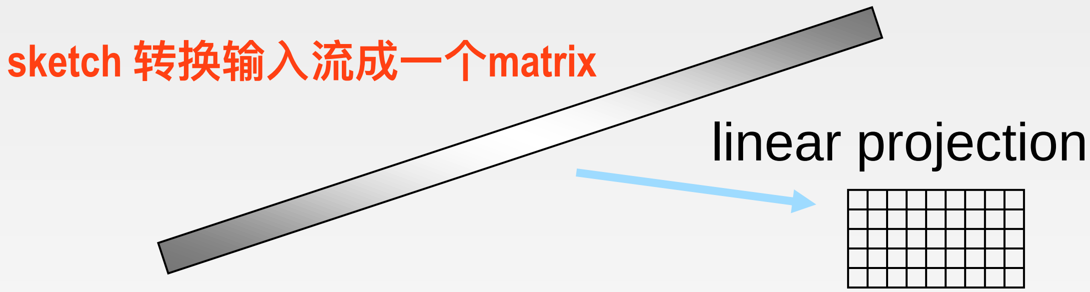
- How many distinct products have we sold in the last week?

Using Small Storage

- Real problem: What if we do not have space to maintain the set of elements seen so far?
- Estimate the count in an unbiased way
- Accept that the count may have a little error, but limit the probability that the error is large

Sketches

- Sampling does not work!
  - If a large fraction of items aren't sampled, don't know if they are all same or all different
- Sketch: a technique takes advantage that the algorithm can "see" all the data even if it can't "remember" it all
- Essentially, sketch is a linear transform of the input
  - Model stream as defining a vector, sketch is result of multiplying stream vector by an (implicit) matrix



Flajolet-Martin Sketch

- Probabilistic Counting Algorithms for Data Base Applications. 1985.

- Pick a hash function ***h*** that maps each of the ***N*** elements to at least $\log_2 N$ bits

- For each stream element ***a***, let ***r*(*a*)** be the number of trailing **0s** in ***h*(*a*)**
  - **r(a)** = position of first 1 counting from the right
    - E.g., say ***h(a) = 12***, then ***12*** is ***1100*** in binary, so ***r(a) = 2***

- Record R = the maximum $r(a)$ seen
  - $\mathbf{R = max_a\ r(a)}$, over all the items ***a*** seen so far

- Estimated number of distinct elements = 2^R

Why It Works: Intuition

- Very very rough and heuristic intuition why Flajolet-Martin works:
  - ***h(a)*** hashes ***a*** with **equal prob.** to any of ***N*** values
  - Then ***h(a)*** is a sequence of $\log_2 N$ bits,
    where 2^{-r} fraction of all ***a***s have a tail of ***r*** zeros
    - About 50% of ***a***s hash to *****0**
    - About 25% of ***a***s hash to ****00**
    - So, if we saw the longest tail of ***r=2*** (i.e., item hash ending ***100**) then we have probably seen **about *4*** distinct items so far
  - So, it takes to hash about 2^r items before we see one with zero-suffix of length r

Why It Works: More formally

- Formally, we will show that **probability of finding a tail of *r* zeros:**
  - **Goes to 1 if** $m \gg 2^r$
  - **Goes to 0 if** $m \ll 2^r$

  where m is the number of distinct elements seen so far in the stream

- Thus, 2^R will almost always be around *m!*

Why It Works: More formally

- **The probability that a given *h*(*a*) ends in at least *r* zeros is 2^{-r}**
  - **h(a)** hashes elements uniformly at random
  - Probability that a random number ends in at least ***r*** zeros is $\mathbf{2^{-r}}$
- Then, the probability of **NOT** seeing a tail of length ***r*** among ***m*** elements:

$$\boxed{\left(\boxed{1-2^{-r}}\right)^{m}}$$

Prob. all end in fewer than ***r*** zeros.

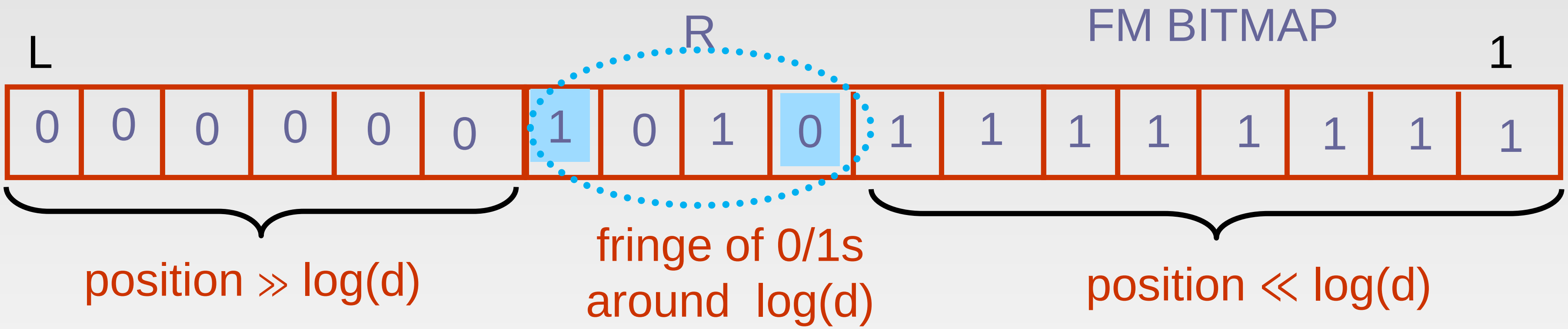
Prob. that given **h(a)** ends in fewer than ***r*** zeros

Why It Works: More formally

- **Note:** $(1-2^{-r})^m = (1-2^{-r})^{2^r(m2^{-r})} \approx e^{-m2^{-r}}$
- **Prob. of NOT finding a tail of length *r* is:**
  - If ***m* << *2^r***, then prob. tends to **1**
    - $(1-2^{-r})^m \approx e^{-m2^{-r}} = 1$ as **$m/2^r \rightarrow 0$**
    - So, the probability of finding a tail of length ***r*** tends to **0**
  - If ***m* >> *2^r***, then prob. tends to **0**
    - $(1-2^{-r})^m \approx e^{-m2^{-r}} = 0$ as **$m/2^r \rightarrow \infty$**
    - So, the probability of finding a tail of length ***r*** tends to **1**
- **Thus, 2^R will almost always be around *m!***

Flajolet-Martin Sketch

- Maintain FM Sketch = bitmap array of L = $\log N$ bits
  - Initialize bitmap to all 0s
  - For each incoming value a, set FM[r(a)] = 1
- If d distinct values, expect d/2 map to FM[1], d/4 to FM[2]…



- Use the leftmost 1: R = $\mathbf{max_a\ r(a)}$
- Use the rightmost 0: also an indicator of log(d)
  - Estimate $d = c2^R$ for scaling constant c ≈ 1.3 (original paper)
- Average many copies (different hash functions) improves accuracy

Part 5: Finding Frequent Elements (Majority and Heavy Hitters)

The Majority Problem

- Given a stream of elements, find the majority if there is one
  - A majority element in the data stream (assume that we have received n elements already) is an element that appears more than n/2 times
- A A B C D B A A B B A A A A A A C C C D A B A A A
  - Answer: A
- It is trivial if we have enough memory
  - For each received element, keep a counter for it. Once receiving it again, increase the counter
  - Can use the binary search tree/hashmap to store the elements
  - O(n log n)/O(n) complexity and O(n) space
- What if we only have limited memory?

Boyer-Moore Voting Algorithm

- This algorithm takes O(n) time and O(1) space
- Basic idea of the algorithm is if we cancel out each occurrence of an element e with all the other elements that are different from e, then e will exist till the end. Then, we can check if it is indeed the majority element.
- Thus, the algorithm contains two phases:
  - First pass: find the possible candidate (the element that has the largest frequency in the stream)
  - Second pass: compute its frequency and verify that it is > n/2

Boyer-Moore Voting Algorithm

- Phase 1:
  - Loop through each element and maintains a count of majority element, and a majority index, maj_index
  - If the next element is same then increment the count, if the next element is not same then decrement the count.
  - if the count reaches 0 then changes the maj_index to the current element and set the count again to 1.

```
maj_index = 0
count = 1
for i in range(len(A)):
    if A[maj_index] == A[i]:
        count += 1
    else:
        count -= 1
    if count == 0:
        maj_index = i
        count = 1
return A[maj_index]
```

Boyer-Moore Voting Algorithm

- Example: given a stream as A[] = 2, 2, 3, 5, 2, 2, 6
  - maj_index = 0, count = 1 –> candidate 2?
  - Same as a[maj_index] => count = 2
  - Different from a[maj_index] => count = 1
  - Different from a[maj_index] => count = 0
  - Since count = 0, change candidate for majority element to 5 => maj_index = 3, count = 1
  - Different from a[maj_index] => count = 0
  - Since count = 0, change candidate for majority element to 2 => maj_index = 4
  - Same as a[maj_index] => count = 2
  - Different from a[maj_index] => count = 1
  - Finally, candidate for majority element is 2

Boyer-Moore Voting Algorithm

- Phase 2: Just compute the count of the element in the stream for verification

```
count = 0
for i in range(len(A)):
    if A[i] == cand:
        count += 1
if count > len(A)/2:
    return True
else:
    return False
```

- We can see that this algorithm still requires two passes of the stream, which is actually not possible in most streaming applications.

- If only one pass and O(1) space allowed, not possible to get the majority element!

Input is an array: https://leetcode.com/problems/majority-element/

Heavy Hitters

- A more general problem: find all elements with counts > n/k (k>=2)
  - There can be at most k such values; and there might be none
  - Trivial if we have enough storage

- Applications
  - Computing popular products. For example, A could be all of the page views of products on amazon.com yesterday. The heavy hitters are then the most frequently viewed products
  - Computing frequent search queries. For example, A could be all of the searches on Google yesterday. The heavy hitters are then searches made most often
  - Identifying heavy TCP flows. Here, A is a list of data packets passing through a network switch, each annotated with a source-destination pair of IP addresses. The heavy hitters are then the flows that are sending the most traffic. This is useful for, among other things, identifying denial-of-service attacks

Approximate Heavy Hitters

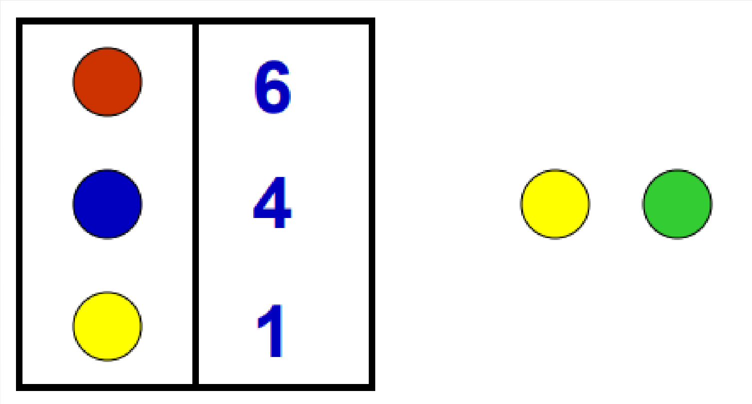
- There is no exact algorithm that solves the Heavy Hitters problems in one pass while using a sublinear amount of auxiliary space

- Relaxation, the ε-approximate heavy hitters problem:
  - If an element has count > n/k, it must be reported, together with its estimated count with (absolute) error < εn
  - If an element has count < (1/k − ε) n, it cannot be reported
  - For elements in between, don't care

- In fact, we will estimate all counts with at most εn error

Misra-Gries Algorithm

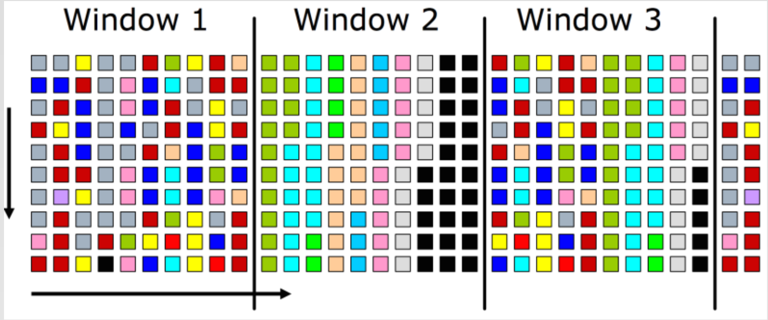
- Keep k different candidates in hand (thus with space O(k))
- For each element in stream:
  - If item is monitored, increase its counter
  - Else, if < k items monitored, add new element with count 1
  - Else, decrease all counts by 1, and delete element with count 0



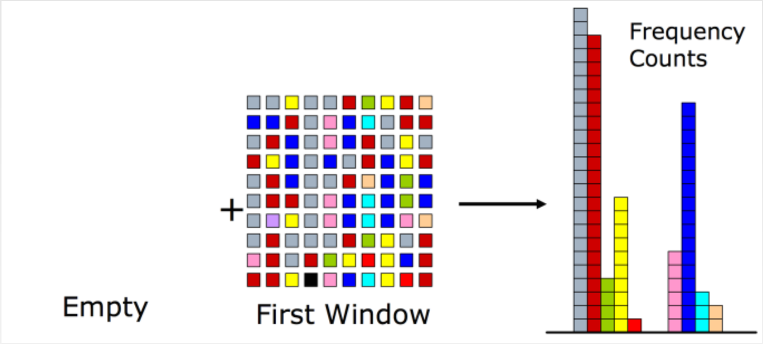
- Each decrease can be charged against k arrivals of different items, so no item with frequency N/k is missed
- But false positive (elements with count smaller than n/k) may appear in the result

Lossy Counting

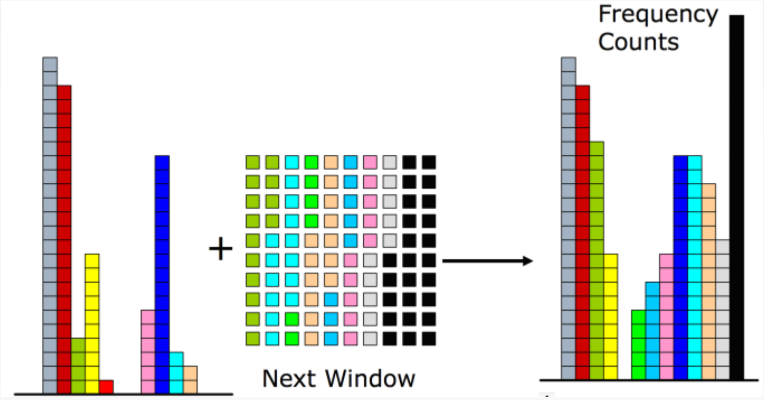
- Step 1: Divide the incoming data stream into windows, and each window contains 1/ε elements



- Step 2: Increment the frequency count of each item according to the new window values. After each window, decrement all counters by 1

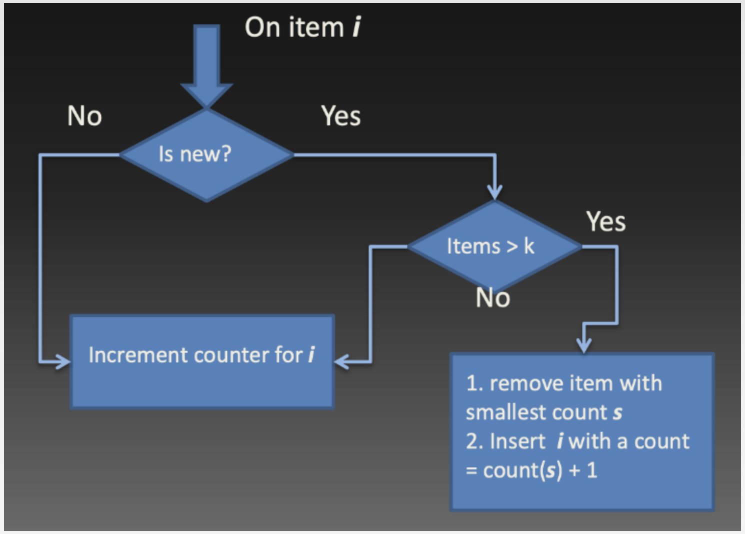


- Step 3: Repeat – Update counters and after each window, decrement all counters by 1



The Space-Saving Algorithm

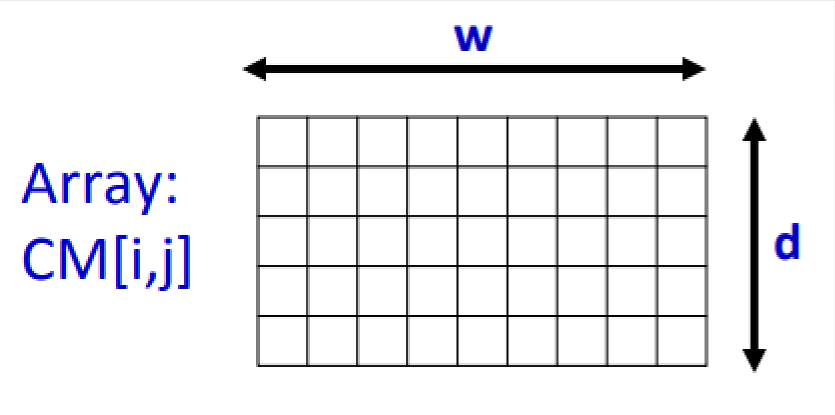
- Keep k = 1/ε item names and counts, initially zero
- On seeing new item:
  - If it has a counter, increment counter
  - If not, replace item with least count, increment count



- Analysis:
  - Smallest counter value, min, is at most εn
  - True count of an uncounted item is between 0 and min
  - Any item x whose true count > εn is stored
- So: Find all items with count > εn, error in counts ≤ εn

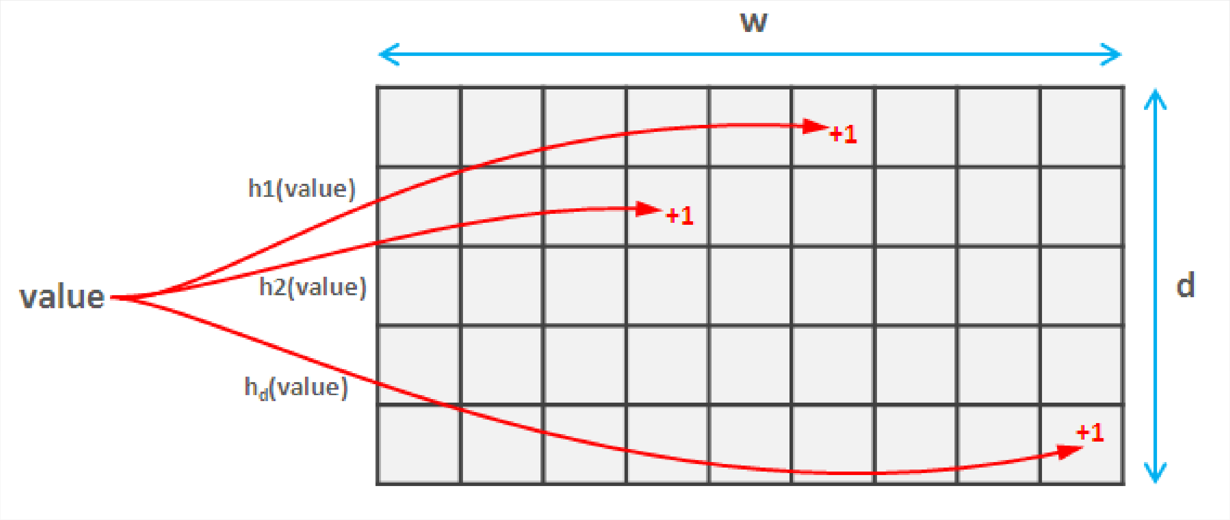
Count-Min Sketch

- In general, model input stream as a vector x of dimension U
  - x[i] is frequency of element I
- The count-min sketch has two parameters, the number of buckets w and the number of hash functions d
- Creates a small summary as an array of w × d in size
- Use d hash function to map vector entries to [1..w]



Count-Min Sketch

- The count-min-sketch supports two operations: Inc(x) and Count(x)
- The operation Count(x) is supposed to return the frequency count of x, meaning the number of times that Inc(x) has been invoked in the past
- The code for Inc(x) is simply:
  - for i = 1, 2, . . . , d: increment CMS[i][hi(x)]



- The code for Count(x) is simply:
  - return $min_{i=1}^{d}$CMS[i][hi(x)]

https://www.geeksforgeeks.org/count-min-sketch-in-java-with-examples/

References

- Chapter 4, Mining of Massive Datasets.
- Finding Frequent Items in Data Streams

End of Chapter 6.2