Week 4: Stream (2)

Instructor: Daejin Choi (djchoi@inu.ac.kr)

INCHEON NATIONAL UNIVERSITY

* Most slides are used from www.mmds.org with small updates

Revisit: Operations on Data Streams

- Stream Processing: choose **a subset of input streams**
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type ***x*** in the last ***k*** elements of the stream
  - **Filtering a data stream**
    - Select elements with property ***x*** from the stream
  - **Counting distinct elements**
    - Number of distinct elements in the last ***k*** elements of the stream
  - **Estimating moments**
    - Estimate avg./std. dev. of last ***k*** elements
  - …

Week 3

Week 4

Filtering a Data Stream: Bloom Filter

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***
  - NOTE: it's different from user-based sampling

- **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

Example Application: Email Spam Filtering

- We know 1 billion "benign" email addresses

- If an email comes from one of these, it is **NOT** spam
  → All benign emails should be determined as benign address
  → Classifying spam as benign is acceptable

Bloom Filter Can Be a Solution

- Bloom Filter consists of
  - An array of bits
  - A number of hash functions

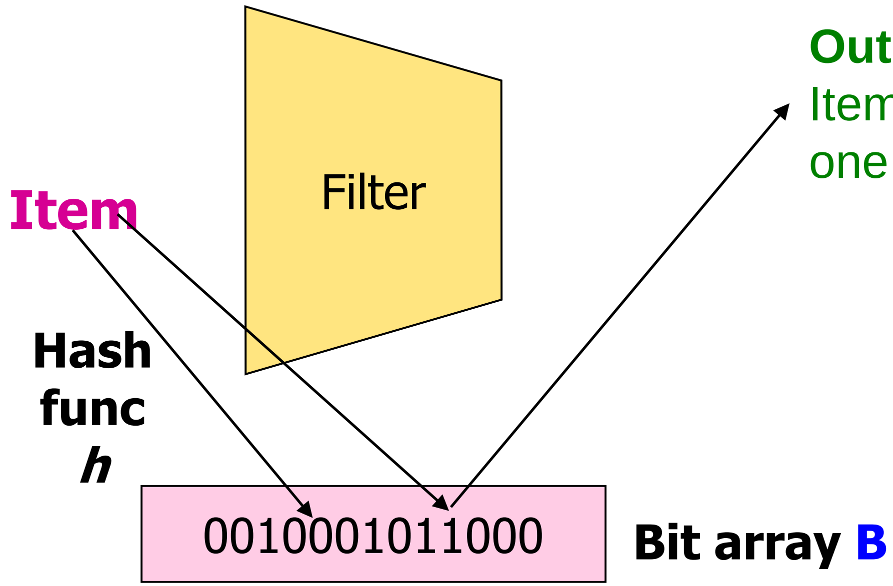
- Two processes
  - Setup process
    - Initially, all bits in the given array set **0**s
    - For each element $\boldsymbol{x}$ in $\boldsymbol{S}$, set the bits $h(\boldsymbol{x})$ to 1, for each hash function $\boldsymbol{h}$
  - Lookup process
    - For input $\boldsymbol{y}$, check if the bits $h(\boldsymbol{y})$ set to 1,
      for each hash function $\boldsymbol{h}$
    - Accept if all bits are set to 1, otherwise reject

Starting from Bloom Filter with 1 Hash

- **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**

Bloom Filter Processing

Output the item since it may be in *S*.
Item hashes to a bucket that at least one of the items in **S** hashed to.



Drop the item.
It hashes to a bucket set to **0** so it is surely not in ***S***.

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

Example

- Use n = **11** bits for our filter.
- Stream elements = integers.
- Use two hash functions:
  - $h_1(x)$ =
    - Take odd-numbered bits from the right in the binary re presentation of x.
    - Treat it as an integer i.
    - Result is i modulo 11.
  - $h_2(x)$ = same, but take **even**-numbered bits.

Example – Continued

Stream element	h_1	h_2	Filter contents
			00000000000
25 = 11001	5	2	00100100000
159 = 10011111	7	0	10100101000
585 = 1001001001	9	7	10100101010

Note: bit 7 was already 1.

인천대학교 INCHEON NATIONAL UNIVERSITY

Example: Lookup

- Suppose we have the same Bloom filter as be fore, and we have set the filter to

  10100101010.
- Lookup element y = 118 = 1110110 (binary).
- $h_1(y)$ = 14 modulo 11 = 3.
- $h_2(y)$ = 5 modulo 11 = 5.
- Bit 5 is 1, but bit 3 is 0, so we are sure y is n ot in the set.

Return to Our Original Question

- **|S| = 1 billion (benign) 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***)

- However, the UNKNOWN email address can be determined a s benign (***false positives***) → We need to estimate

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** = input 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

$$\boxed{1 - \boxed{\underline{(1 - 1/n)}^{\,n}}^{(m/n)}}$$

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

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

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

Analysis: Throwing Darts (3)

- **Fraction of 1s (over input) in the array B =probability of false positive = $1 - e^{-m/n}$**

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

Bloom Filter with More Hash Functions

- Consider: **|S| = *m*, |B| = *n***
- Use ***k*** independent hash functions ***h_1,…, h_k***
- **Initialization:**
  - Set **B** to all **0s** (**note:** we have a single array B!)
  - Hash each element ***s* ∈ *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** **for all** ***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 – Analysis

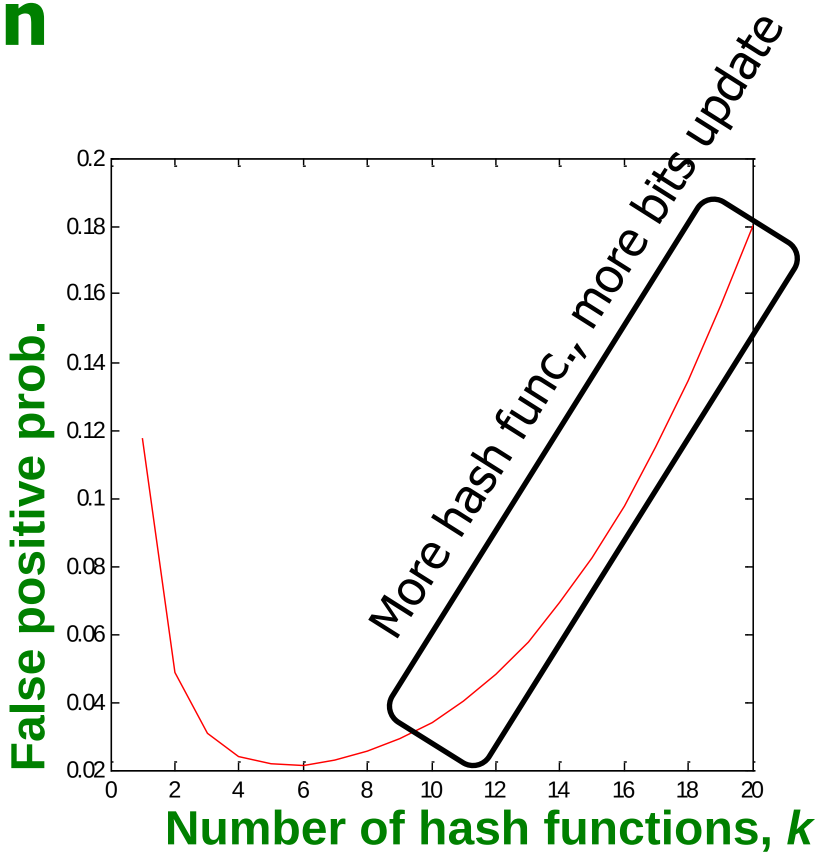
- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of **1**s is $(1 - e^{-km/n})$

- 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**: $(1 - e^{-1/6})^2$ = **0.0235**

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

Counting Distinct Elements

Flajolet-Martin Approach

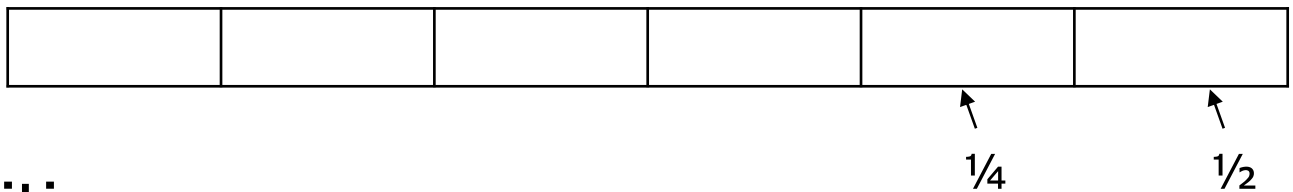
- Pick a hash function ***h*** that maps each of the ***N*** elements to at least $\log_2 N$ bits

- 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

- Hash each item x to a bit, using exponential distribution
  - ½ map to bit 0, ¼ map to bit 1, …



- Intuition
  - The 0th bit is accessed with prob. ½
  - The 1st bit is accessed with prob. ¼
  - …
  - The kth bit is accessed with prob. O($1/2^k$)
- Thus, if the kth bit is set, then we know that an event with prob. O($1/2^k$) happened

  → We inserted distinct items O(2^k) times

Why It Works: More formally

- **Goal: showing 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

→ The goal also means that 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:

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

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/2ʳ→ 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/2ʳ → ∞**
    - So, the probability of finding a tail of length ***r*** tends to **1**

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

Computing Moments

Generalization: Moments

- **Suppose a stream has elements chosen from a set *A* of *N* values**

- **Let m_i be the number of times value *i* occurs in the stream**

- **The k^{th} *moment* (적률) is**

$$\sum_{i \in A} (m_i)^k$$

Special Cases

$$\sum_{i \in A} (m_i)^k$$

- **0th moment =** number of distinct elements
  - The problem just considered
- **1st moment =** count of the numbers of elements = length of the stream
  - Easy to compute
- **2nd moment =** ***surprise number S*** **=** a measure of how uneven the distribution is

Example: Surprise Number

- **Stream of length 100**
- **11 distinct values**

- Item counts: **10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9**
  Surprise *S* = 910

- Item counts: **90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1**
  Surprise *S* = 8,110

Problem Definition

- **Q**: Given a stream, how can we estimate k-th moments efficiently, with small memory space?

- **A**: AMS method

AMS Method

[Alon, Matias, and Szegedy]

- **AMS method works for all moments**
- **Gives an unbiased estimate**
- **We will just concentrate on the 2nd moment *S***
- **We pick and keep track of many variables *X*:**
  - For each variable ***X*** we store ***X.el*** and ***X.val***
    - ***X.el*** corresponds to the item ***i***
    - ***X.val*** corresponds to the **count** of item ***i***
  - Note this requires a count in main memory, so number of ***X***s is limited
- **Our goal is to compute** $S = \sum_i m_i^2$

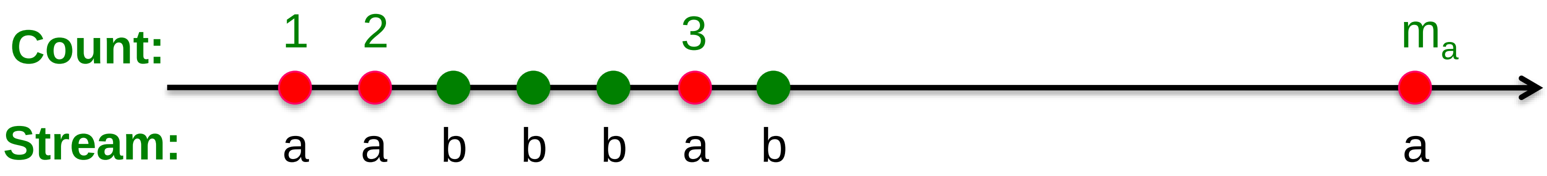
- **How to set *X.val* and *X.el*?**
  - Assume stream has length ***n*** (we relax this later)
  - Pick some random time ***t*** (***t<n***) to start, so that any time is equally likely
  - Let at time ***t*** the stream have item ***i***. ***We set X.el = i***
  - Then we maintain count ***c*** (***X.val = c***) of the number of ***i***s in the stream starting from the chosen time ***t***

- **Then the estimate of the 2nd moment ($\sum_i m_i^2$) is:**

$$S = f(X) = n\,(2 \cdot c - 1)$$

  - Note, we will keep track of multiple **X**s, ($X_1, X_2, \ldots X_k$) and our final estimate will be $S = 1/k \sum_j^k f(X_j)$

Expectation Analysis



- **2nd moment is** $S = \sum_i m_i^2$
- c_t … number of times item at time t appears from time **t** onwards (c_1=1 , c_2=2, c_3=1)
- $E[f(X)] = \frac{1}{n}\sum_{t=1}^{n} n(2c_t - 1)$

$$= \frac{1}{n}\sum_i n\,(1 + 3 + 5 + \cdots + 2m_i - 1)$$

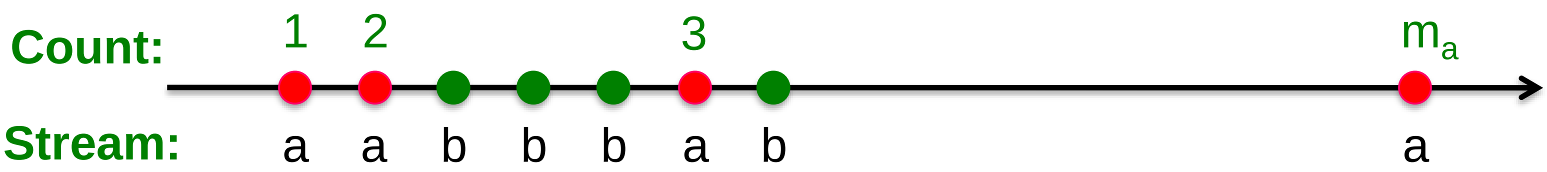
m_i … total count of item i in the stream (we are assuming stream has length **n**)

Group times by the value seen

Time t when the last i is seen (c_t=1)

Time t when the penultimate i is seen (c_t=2)

Time t when the first i is seen (c_t=m_i)



- $E[f(X)] = \frac{1}{n}\sum_i n\,(1 + 3 + 5 + \cdots + 2m_i - 1)$
    - Little side calculation: $(1 + 3 + 5 + \cdots + 2m_i - 1) = \sum_{i=1}^{m_i}(2i - 1) = 2\frac{m_i(m_i+1)}{2} - m_i = (m_i)^2$
- **Then** $\boldsymbol{E[f(X)] = \frac{1}{n}\sum_i n\,(m_i)^2}$
- **So,** $\mathbf{E[f(X)]} = \sum_i (m_i)^2 = S$
- **We have the second moment (in expectation)!**

Higher-Order Moments

- **For estimating kth moment we essentially use the same algorithm but change the estimate:**
  - For **k=2** we used ***n*** **(2·c – 1)**
  - For **k=3** we use: ***n*** **(3·c^2 – 3c + 1)** (where **c=X.val**)
- **Why?**
  - **For k=2:** Remember we had $(1 + 3 + 5 + \cdots + 2m_i - 1)$ and we showed terms ***2c-1*** (for **c=1,…,m**) sum to ***m²***
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c-1)^2 = m^2$
    - **So:** $2c - 1 = c^2 - (c-1)^2$
  - **For k=3:** **c^3 - (c-1)^3 = 3c^2 - 3c + 1**
- **Generally:** Estimate $= n\,(c^k - (c-1)^k)$

Combining Samples

- **In practice:**
  - Compute $f(X) = n(2c - 1)$ for as many variables X as you can fit in memory
  - Average them in groups
  - Take median of averages

- **Problem: Streams never end**
  - We assumed there was a number n, the number of positions in the stream
  - But real streams go on forever, so n is a variable – the number of inputs seen so far

Streams Never End: Fixups

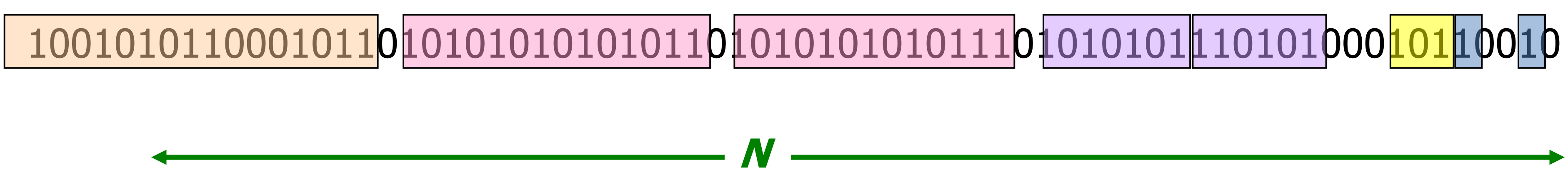
- The variables ***X*** have ***n*** as a factor – keep ***n*** separately; just hold the count in ***X***
- Suppose we can only store ***k*** counts. We must throw some ***X***s out as time goes on:
  - **Objective:** Each starting time ***t*** is selected with probability ***k/n***
  - **Solution: (fixed-size sampling!)**
    - Choose the first ***k*** times for ***k*** variables
    - When the ***n*th** element arrives (***n* > *k***), choose it with probability ***k/n***
    - If you choose it, throw one of the previously stored variables **X** out, with equal probability

Counting Itemsets

Counting Itemsets

- **New Problem:** **Given a stream, which items appear more than *s* times in the window?**

- **Possible solution:** Think of the stream of baskets as one binary stream per item
  - **1** = item present; **0** = not present
  - Use **DGIM** to estimate counts of **1**s for all items



Extensions

- **In principle, you could count frequent pairs or even larger sets the same way**
  - **One stream per itemset**
  - E.g., for a basket {i, j, k}, assume 7 independent streams: (i) (j) (k) (i, j) (i, k) (j, k) (i, j, k)

- **Drawbacks:**
  - **Number of itemsets is way too big**

Exponentially Decaying Windows

- **Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)**
  - **What are "currently" most popular movies?**
    - Instead of computing the raw count in last ***N*** elements
    - Compute a **smooth aggregation** over the whole stream

- If stream is $a_1, a_2, \ldots$ and we are taking the sum of the stream, take the answer at time t to be: $= \sum_{i=1}^{t} a_i (1-c)^{t-i}$
    - **c** is a constant, presumably tiny, like $\mathbf{10^{-6}}$ or $\mathbf{10^{-9}}$

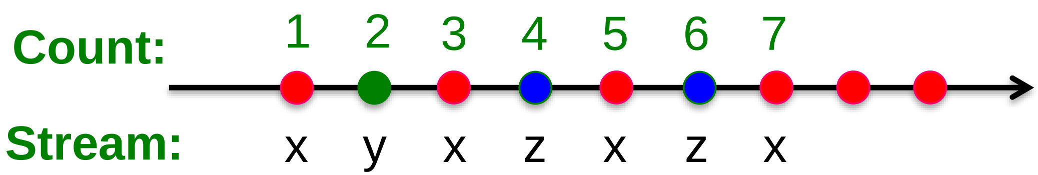
- **When new a_{t+1} arrives:**
  Multiply current sum by **(1-c)** and add $\mathbf{a_{t+1}}$

Example: Counting Items

- If each a_i is an "item" we can compute the **characteristic function** of each possible item x as an Exponentially Decaying Window
  - That is: $\sum_{i=1}^{t} \delta_i \cdot (1-c)^{t-i}$
    where $\delta_i = 1$ if $a_i = x$, and **0** otherwise
  - Imagine that for each item x we have a binary stream (**1** if x appears, **0** if x does not appear)
  - New item x arrives:
    - Multiply all counts by **(1-c)**
    - Add **+1** to count for element x

- **Call this sum the "weight" of item *x***

Example: Counting Items



$$\sum_{i=1}^{t} \delta_i \cdot (1-c)^{t-i}$$

Assume c = 0.2, Keep items with weights >= 1/2

- (T1) x: 1
- (T2) x: 0.8*1, y: 1
- (T3) x: 0.8*0.8 + 1, y: 0.8*1
- (T4) x: 0.8*1.64, y: 0.8*0.8, z = 1
- (T5) x: 1.312+1, y: 0.8*0.64=0.512, z: 0.8*1
- (T6) x: 0.8*2.312, y: 0.8*0.512, z:0.8*0.8
  - Remove y
- (T7) x: 0.8*1.8496+1, z: 0.8*0.64
- …

Example: Counting Items

- **What are "currently" most popular movies?**
- **Suppose we want to find movies of weight > ½**
  - **Important property:** Sum over all weights $\sum_t (1-c)^t$ is 1/[1 – (1 – c)] **= 1/c**
- **Thus:**
  - There cannot be more than **2/c** movies with weight of **½**
- So, **2/c** is a limit on the number of movies being counted at any time

Extension to Itemsets

- **Count (some) itemsets**
  - **What are currently "hot" itemsets?**
  - **Problem:** Too many itemsets to keep counts of all of them in memory

- **When a basket B comes in:**
  - Multiply all counts by **(1-c)**
  - For uncounted items in **B**, create new count
  - Add **1** to count of any item in **B** and to any **itemset** contained in **B** that is already being counted
  - **Drop counts < ½**
  - Initiate new counts (next slide)

Initiation of New Counts

- Start a count for an itemset $S \subseteq B$ if every proper subset of S had a count prior to arrival of basket B
  - **Intuitively:** If all subsets of **S** are being counted this means they are "**frequent/hot**" and thus ***S*** has a potential to be "**hot**"
- **Example:**
  - Start counting ***S*={i, j}** iff both **i** and **j** were counted prior to seeing ***B***
  - Start counting ***S*={i, j, k}** iff **{i, j}**, **{i, k}**, and **{j, k}** were all counted prior to seeing ***B***

Summary

- **Filtering a data stream:** Bloom Filter
- **Counting distinct elements:** Flajolet-Martin
- **Computing moments**: AMS Method
- **Counting itemsets**: Exponentially Decaying Windows

Intermediate Summary

- We have covered **basic operations** for **big data processing**
  - How to deal with **large (static) data**? MapReduce
  - How to deal with **stream data**?
    - Sampling, sliding window, …
    - Estimation + summarized results

- In practice,
  - Processing frameworks are well developed (& easy to use!)
  - The opportunities for you to develop operations from the beginning are rare
  - However, understanding the problems and concept of solutions will be helpful to design

- Now we are moving to **Data Analysis**

Thank you!

Instructor: Daejin Choi (djchoi@inu.ac.kr)