Mining Data Streams

Professor Wei-Min Shen
University of Southern California

Thanks for source slides and material to:
J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets
http://www.mmds.org

Mining Data Stream

- Motivation
- Sampling
  - Fixed-portion sampling
  - Fixed-size (reservoir) sampling
- Filtering
  - Bloom filter
- Counting
  - Estimating # of distinct values, moments
- Sliding window
  - Counting # of 1's in the window

Data streams & applications

- Query streams
  - How many unique users at Google last month?
- URL streams while crawling
  - Which URLs have been crawled before?
- Sensor data
  - What is the maximum temperature so far?

Stream data processing

- Stream of tuples arriving at a rapid rate
  – In contrast to traditional DBMS where all tuples are stored in secondary storage

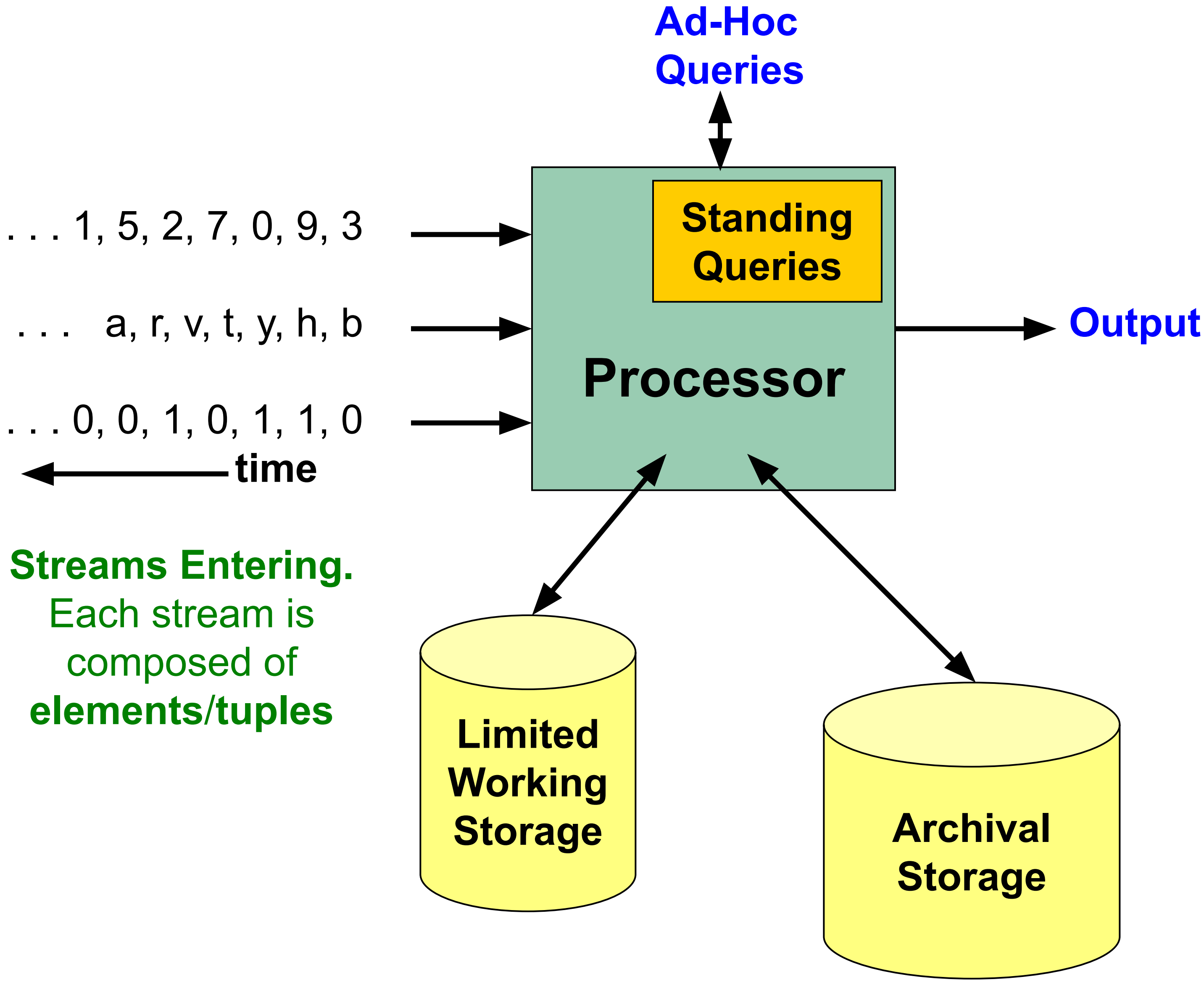
- Infeasible to use all tuples to answer queries
  – Cannot store them all in main memory
  – Too much computation
  – Query response time critical

Query types

- Standing queries
  - Executed whenever a new data tuple arrives
  - keep only one value
    - e.g., report each new maximum value ever seen in the stream

- Ad-hoc queries
  - Normal queries asked one time
  - Need entire stream to have an exact answer
    - e.g., what is the number of unique visitors in the last two months?

Stream Processing Model



Example: Running averages

- Given a window of size *N*
  - Report the average of values in the window whenever a value arrives
  - *N* is so large that we can not store all tuples in the window

- What is the type of this query?
- How to do this?

Example: Running averages

- First *N* inputs, accumulate sum and count
  - Avg = sum/count

- A new element *i*
  - Change the average by adding (i - j)/N
    - *j* is the oldest element in the window
    - window size is fixed so we need to discard *j*

Roadmap

- Motivation
- Sampling
  - Fixed-portion & fixed-size (reservoir sampling)
- Filtering
  - Bloom filter
- Counting
  - Estimating # of distinct values, moments
- Sliding window
  - Counting # of 1's in the window

Sampling from a data stream

- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - Answer questions such as: How often did a user run the same query in a single day
  - Have space to store **1/10th** of query stream

- Method 1: sample a fixed portion of elements
  - e.g., 1/10

- Method 2: maintain a fixed-size sample

SAMPLING
FIXED-PORTION

Sampling a fixed proportion

- Search engine query stream
  - Stream of tuples: (user, query, time)
- Example query
  - What fraction of queries by a user are duplicates?
- Assumption
  - Have space to store 1/10 of stream tuples

Naive solution

- For each tuple (store it or not?)
  - generate a random number in [0..9]
  - store it in the sample if the number is 0
    - Sample rate?
- Example stream tuples:
  - (john, data mining, 2015/12/01 9:45)
  - (mary, inf 553, 2015/12/01 10:08)
  - (john, data mining, 2015/12/01 11:30)
  - …
- Problems?
  - Your sampling may contain duplicates

The true fraction of queries with duplicates

- A user issued ***s*** queries once & ***d*** queries twice
  - E.g., data mining, inf 553, movie, movie, tom, tom (s=d=2)
- Total number of queries & duplicates = s + 2d
- True fraction of queries with duplicates = $\boxed{d/(s+d)}$
- Sampling rate = 1/10

Queries with duplicates in sample

- The sample contains:
  - s/10 "***s***" queries, e.g., data mining
  - 2d/10 "***d***" tuples, e.g., movie, tom, tom
- If sample = data mining, movie, tom, tom, question:
  - What is the expected number of ***d*** queries with duplicates in the sample?
    - movie – ***d*** query without duplicates in the sample
    - tom, tom – ***d*** query with duplicates in the sample
  - E.g., both tom's appear in the sample

Queries with duplicates in sample

- $s_1, s_2, \ldots, s_{800}, d_1, d_1', d_2, d_2', \ldots, d_{100}, d_{100}'$
  - $s = 800$, $d = 100$

- Each $\boldsymbol{d_j}$ has probability of 1/10 being selected
  - so prob. of two $\boldsymbol{d_j}$'s being selected = 1/10 * 1/10
- There are $\boldsymbol{d}$ number of $\boldsymbol{d_j}$'s

⇒ so the expected number of <u>duplicated pairs</u> in sample = d/100

⇒ or d/100 queries + d/100 their duplicates

  ⇒ That is 2d/100

“d” Queries without duplicates in sample

- The sample contains:
  - s/10 “***s***” queries, e.g., data mining
  - 2d/10 “***d***” tuples, e.g., movie, tom, tom

- Question:
  - What is the expected number of ***d*** queries without duplicates in the sample?
  - E.g., only one movie appears in the sample

“d” Queries without duplicates in sample

- $s_1, s_2, \ldots, s_{800}, d_1, d_1', d_2, d_2', \ldots, d_{100}, d_{100}'$
  - $s = 800$, $d = 100$

- Expected # of singleton d queries in sample
  - $\boldsymbol{d_j}$ selected, $\boldsymbol{d_j'}$ not selected: 1/10 * 9/10
  - $\boldsymbol{d_j}$ not selected, $\boldsymbol{d_j'}$ selected: 9/10 * 1/10

  => 9d/100+9d/100 = 18d/100

Fraction of queries in sample w/ duplicates

- $s_1, s_2, \ldots, s_{800}, d_1, d_1', d_2, d_2', \ldots, d_{100}, d_{100}'$
  - s = 800, d = 100
- Fraction $= \dfrac{\frac{d}{100}}{\frac{s}{10}+\frac{d}{100}+\frac{18d}{100}} = \dfrac{d}{10s+19d} \neq \dfrac{d}{s+d}$
- Note total # of ***d*** queries with and without duplicates = d/100 * 2 + 18d/100
  = 20d/100
  = 2d/10

What has been the problem?

- Mistake: sample by position
  - When search tuple arrives, we flip a 10-sided dice
  - Retain it if the dice shows up as 0

- Solution: sample by user (by key)
  - When search tuple (user, query, time) arrives
  - We extract its user (key) component
  - Hash user into 10 buckets: 0, 1, …, 9
  - Retain all tuples for the user in the bucket 0

Sample by user

- Sample = all queries for users hashed into bucket 0

- All or none of queries of a user will be selected

- Thus, the fraction of unique queries in sample
  - will be the same as that in the stream as a whole

General Sampling Problem

- Stream of tuples with ***n*** components
  - Key = a subset of components
- Search stream: (user, query, time)
  - Sample size: a/b
- Sampling strategy:
  - Hash key (e.g., user) to ***b*** buckets
  - Accept tuple if key value < a

Example

- Tuples: (empID, dept, salary)
- Query: avg. range of salary within a dept.?

- Randomly selecting tuples
  - Might miss the min/max salary

- Key = dept.
- Sample: some departments and all tuples in these departments

SAMPLING
FIXED-SIZED (RESERVOIR)

Roadmap

- Motivation
- Sampling
  - Fixed-portion sampling
  - Fixed-size (reservoir) sampling
- Filtering
  - Bloom filter
- Counting
  - Estimating # of distinct values, moments
- Sliding window
  - Counting # of 1's in the window

Problem with fixed portion sample

- Sample size may grow too big when data stream in
  - Even 10% could be too big, e.g., tuples in bucket 1/10 exceed the memory size

- Idea: throw away some queries

- Key: do this consistently
  - remove all or none of occurrences of a query

Controlling the sample size

- Put an upper bound on the sample size
  - Start out with 10%

- Solution:
  - Hash queries to a large # of buckets, say 100
  - Put them into sample if they hash to bucket 0 to 9
  - When sample grows too big, throw away bucket 9
  - So on

Maintaining a fixed-size sample

- Suppose we need to maintain a random sample, ***S***, of size exactly *s* tuples (instead of %)
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance
- Suppose at time ***n*** we have seen ***n*** items
  - Each item is in the sample ***S*** with equal prob. ***s/n***
    - A challenge to achieve !

How to think about the problem: say s = 2

Stream: a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.

At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.

Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the ***n****th* element arrives (***n* > *s***)
    - With probability ***s/n***, keep the ***n****th* element, else discard it
    - If we picked the ***n****th* element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random
- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element seen so far with probability ***s/n***

Proof: By Induction

- **We prove this by induction:**
  - Assume that after ***n*** elements, the sample contains each element seen so far with probability ***s/n***
  - We need to show that after seeing element ***n+1*** the sample maintains the property
    - Sample contains each element seen so far with probability ***s/(n+1)***
- **Base case:**
  - After we see **n=s** elements the sample **S** has the desired property
    - Each out of **n=s** elements is in the sample with probability ***s/s = 1***

Proof: By Induction

- **Inductive hypothesis:** After ***n*** elements, the sample ***S*** contains each element seen so far with prob. ***s/n***
- **Now element *n+1* arrives**
- **Inductive step:** For each element x already in ***S***, probability that the algorithm keeps x in ***S*** is:

$$\underset{\text{Element } \mathbf{n+1} \text{ discarded}}{\left(1-\frac{s}{n+1}\right)} + \underset{\text{Element } \mathbf{n+1} \text{ not discarded}}{\left(\frac{s}{n+1}\right)}\underset{\text{Element in the sample not picked}}{\left(\frac{s-1}{s}\right)} = \frac{n}{n+1}$$

- So, at time ***n,*** tuples in ***S*** were there with prob. **s/n**
- Time ***n→n+1,*** tuple stayed in ***S*** with prob. **n/(n+1)**
- So prob. tuple is in ***S*** at time ***n+1*** **=** $\frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Roadmap

- Motivation
- Sampling
  - Fixed-portion & fixed-size (reservoir sampling)
- Filtering
  - Bloom filter
- Counting
  - Estimating # of distinct values, moments
- Sliding window
  - Counting # of 1's in the window

FILTERING

Stream Filtering

- Application: spam filtering
  - Check incoming emails against a large set of known email addresses (e.g., 1 billion)

- Application: URL filtering in Web crawling
  - Check if discovered URL's have already been crawled

Bloom filter

- Check if an object ***o*** is in a set ***S***
  - without comparing ***o*** with all objects in ***S*** (explicitly)

- No false negatives
  - If Bloom says no, then ***o*** is definitely not in ***S***

- But may have false positives
  - If Bloom says yes, <u>it is possible</u> that ***o*** is not in ***S***

Components in a Bloom filter

- An array A of n bits, initially all 0's: $A[0..n-1]$
- A set of hash functions, each
  - takes an object (stream element) as the input
  - returns a position in the array: $0..n-1$
- A set S of objects

Construct and apply the filter

- Construction: for each object $\boldsymbol{o}$ in $\boldsymbol{S}$,
  - Apply each hash function $\boldsymbol{h_j}$ to $\boldsymbol{o}$
  - If $\boldsymbol{h_j(o)} = \boldsymbol{i}$, set $\boldsymbol{A[i]} = \boldsymbol{1}$ (if it was 0)

- Application: check if new object $\boldsymbol{o}$' is in $\boldsymbol{S}$
  - Hash $\boldsymbol{o}$' using each hash function
  - If for some hash function $\boldsymbol{h_j(o') = i}$ and $\boldsymbol{A[i] = 0}$, stop and report $\boldsymbol{o'}$ not in $\boldsymbol{S}$

Example

- 11-bit array (n=11)
- Stream elements = integers
- Two hash functions
  - $h_1(x)$ = (odd-position bits from the right) % 11
  - $h_2(x)$ = (even-position bits from the right) % 11

Example: Building the filter

Stream element	h_1	h_2	Filter
			0 0 0 0 0 0 0 0 0 0 0
25 = 1 1 0 0 1	5	2	0 0 1 0 0 1 0 0 0 0 0
159 = 1 0 0 1 1 1 1 1	7	0	1 0 1 0 0 1 0 1 0 0 0
585 = 1 0 0 1 0 0 1 0 0 1	9	7	1 0 1 0 0 1 0 1 0 1 0

Example: Using the filter

- Is 118 in the set?
  - No false negative in Bloom

Stream element	h_1	h_2
118 = 1 1 1 0 1 1 0	3	5

Filter

1	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**

Example: False positive

- Is 38 in the set (25, 159, 585)?
  - It turns on the same bits as 25, but in diff. ways

Stream element	h_1	h_2	Filter
38 = 1 0 0 1 1 0	2	5	1 0 1 0 0 1 0 1 0 1 0

Recall

25 = 1 1 0 0 1	5	2	0 0 1 0 0 1 0 0 0 0 0

False positive

- ***x*** not in ***S***, but identified as in ***S***

- Reason:
  - For all hash functions, ***x*** hashes into an ***1*** position
  - That is, $h_i(x) = h_j(e)$, for some ***e*** in ***S***
  - Note: ***j*** may be different from ***i***

False positive rate (upper bound)

- ***n*** = # of bits in array
- ***k*** = # of hash functions
- ***m*** = # of elements inserted

- ***f*** = fraction of 1's in bit array
- False positive rate = $\boldsymbol{f^{k}}$
  – The probability of saying YES to the question "Is X in the set?"

- $\boldsymbol{f \leq m*k/n}$ (this is an upper bound, why?)

Example (upper bound)

- $\boldsymbol{n}$ = 8 billions (bits in array)
- $\boldsymbol{m}$ = 1 billion (objects in the set)
- $\boldsymbol{k}$ = 1 (# of hash function)

- $\boldsymbol{f}$ is estimated to be $\boldsymbol{km/n}$ = 1/8
  - 1/8 of bits in the array are 1

- False positive rate $\leq 1/8 = .125$

Accurate Estimation of fraction of 1's

- n = # of bits in array
- k = # of hash functions
- m = # of elements inserted

- Fraction of 1's = the probability that a bit in the array is set to 1 by at least one hashing
  - Total # of hashings: ***k * m***

Estimation model

- Consider throwing ***d*** darts on ***t*** targets

- What is the probability, denoted as ***p***, of a given target hit by at least one dart?
  - Prob. of a target not hit by **a dart** = ***1-1/t***
  - Prob. of a target not hit by all darts = $\boldsymbol{(1-1/t)^d}$

    since $(1-1/t)^t = ((1+\frac{1}{-t})^{-t})^{-1}) = e^{-1} = 1/e$ for large t
  - we have $(1-1/t)^{t*d/t} = \boldsymbol{e^{-d/t}}$
  - $\boldsymbol{p = 1 - e^{-d/t}}$

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

Estimating the fraction of 1's

- n = # of bits in array
- k = # of hash functions
- m = # of elements inserted

- Fraction of 1's = the probability that a bit in the array is set to 1 by at least one hashing
  - ***$1 - e^{-km/n}$***

False Positive Rate (Accurate)

- f = fraction of 1's in bit array
- k = # of hash functions
- m = # of elements inserted

- False positive rate = $\boldsymbol{f}^{\,\boldsymbol{k}}$

- Instead of $\boldsymbol{f \leq m*k/n}$, we have $\boldsymbol{f = 1 - e^{-km/n}}$
- so false positive rate = $\boldsymbol{(1 - e^{-km/n})^k}$
  - Multiple hash functions hit the same bit

Example (actual rate)

- n = 8 billions (bits in array)
- m = 1 billion (objects in the set)
- k = 1 (# of hash function)

- Recall that false positive rate $\leq 1/8 = .125$
- Actual rate = $\boldsymbol{(1 - e^{-km/n})^k} = \boldsymbol{1\text{-}e^{-1/8}} = .1175$

- What if $k = 2$?

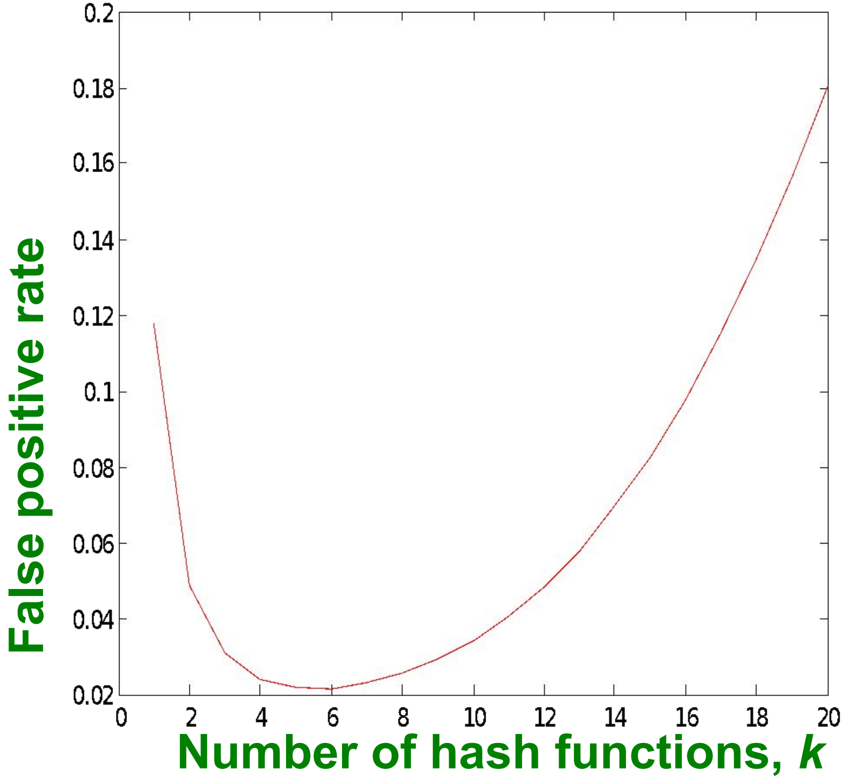
Example (actual rate)

- ***n*** = 8 billions (bits in array)
- ***m*** = 1 billion (objects in the set)
- ***k*** = 2 (# of hash function)

- Actual rate = $\boldsymbol{(1 - e^{-km/n})^k}$ = $(1\text{-}e^{-2/8})^2$ = $.2212^2$ = .0489

Optimal k

- ***n*** = 8 billions
- ***m*** = 1 billion
- ***k*** = # of hash functions



- Rate = $\boldsymbol{(1 - e^{-km/n})^k}$
  $= (1 - e^{-k/8})^k$
- Optimal k = n/m ln(2)
  - E.g., k = 8 ln (2) = 5.54 ~ 6
- Error rate at k = 6: $(1\text{-}e^{-6/8})^6$ = .0216

Roadmap

- Motivation
- Sampling
  - Fixed-portion & fixed-size (reservoir sampling)
- Filtering
  - Bloom filter
- Counting
  - Estimating # of distinct values, moments
- Sliding window
  - Counting # of 1's in the window

COUNTING
(DISTINCT ELEMENTS)

Compute # of distinct elements

- Applications
  - Compute # of distinct users to Facebook
  - Compute # of distinct queries submitted to Google

- Obvious solution:
  - Hash table of distinct elements
  - Check if new element is there; if not, add it
- Problems?

Flajolet-Martin algorithm

- Estimating the counts

1. Hash every element a to a sufficiently long bit-string (e.g., h(element *a*) = 1100 – 4 bits)

2. Maintain R = length of *longest* trailing zeros among all bit-strings (e.g., R = 2)

3. Estimate count = 2^R, e.g., $2^2 = 4$

Example (estimate by trailing 0s)

- Consider 4 distinct elements: a, b, c, d
- Hash value into bit string of length 4
- How likely do we see at least one hash value with a 0 in the last bit? Next slide
  - hash(a) = 0010
  - hash(b) = 0111
  - hash(c) = 1010
  - hash(d) = 1111

Example: at least one ends with 0

- E.g.,
  - hash(a) = 0010
  - hash(b) = 0111
  - hash(c) = 1010
  - hash(d) = 1111
- Prob. of none of hash values ending with 0:
  - ***$(1-½)^4$ (every string ends with 1; there are four strings)***
- Prob. of at least one ending with 0:
  - ***$1-(1-½)^4$= .9375***

Example: at least one ends with 00

- E.g.,
  - hash(a) = 0100
  - hash(b) = 0111
  - hash(c) = 1010
  - hash(d) = 1111

- Prob. of someone ending with 00:
  - $(½)\ (½) = (½)^2 = 2^{-2}$
- Prob. of none ending with 00:
  - $(1-(½)^2)^4=.32$
- Prob. of at least one ending with 00:
  - 1-.32 = .68

Example: at least one ends with 000

- E.g.,
  - hash(a) = 1000
  - hash(b) = 0111
  - hash(c) = 1010
  - hash(d) = 1111

- Prob. of someone ending with 000:
  - $(\frac{1}{2})\ (\frac{1}{2})\ (\frac{1}{2}) = (\frac{1}{2})^3 = 2^{-3}$
- Prob. of none ending with 000:
  - $(1-(\frac{1}{2})^3)^4 = .59$
- Prob. of at least one ending with 000:
  - $1 - .59 = .41$

Why it works?

- Prob. of ***h(a)*** having at least ***r*** trailing 0's
  - 2^{-r}
- Suppose there are ***m*** distinct elements (the true answer)

- Prob. that ***h(a)*** for some element ***a*** has at least ***r*** trailing 0's (as in the examples in the previous slides)

$$1-(1-2^{-r})^m = 1-(1-2^{-r})^{2^r\frac{m}{2^r}} = \mathbf{1-e^{-\frac{m}{2^r}}}$$

Why it works?

- Prob. that ***h(a)*** for some element ***a*** has at least ***r*** trailing 0's

$$p = 1 - (1 - 2^{-r})^m = \mathbf{1 - \ e^{-\frac{m}{2^r}}}$$

- Tayler expansion of e^x

$$1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \sim \mathbf{1 + \frac{x^1}{1!}, \mathit{if}\ |X| \ll 1}$$

- $p = 1 - \ e^{-\frac{m}{2^r}} \sim 1 - \left(1 - \frac{m}{2^r}\right) = \mathbf{\frac{m}{2^r}}$, if $2^r \gg$ m

Why it works?

- Prob. that ***h(a)*** for some element a has at least r trailing 0's
  - $p = 1 - (1 - 2^{-r})^m = 1 - e^{-\frac{m}{2^r}}$
- Prob. that ***h(a)*** for NO element a has at least r trailing 0's
  - $p' = (1 - 2^{-r})^m = e^{-\frac{m}{2^r}}$

(1) If $2^r \gg \mathrm{m}, p = \frac{m}{2^r} \to 0$; $p' = 1$

- the probability of finding a tail length at least r approaches 0
- ***R*** is unlikely to be too large, since otherwise it may not show up at all

(2) If $2^r \ll \mathrm{m}, p = 1 - 1/e^{\frac{m}{2^r}} \to 1$; $p' = 0$

- the probability that we shall find a tail of length at least r approaches 1
- Since ***R*** is the longest 0's, it is unlikely to be too small.

Therefore, 2^R is around m

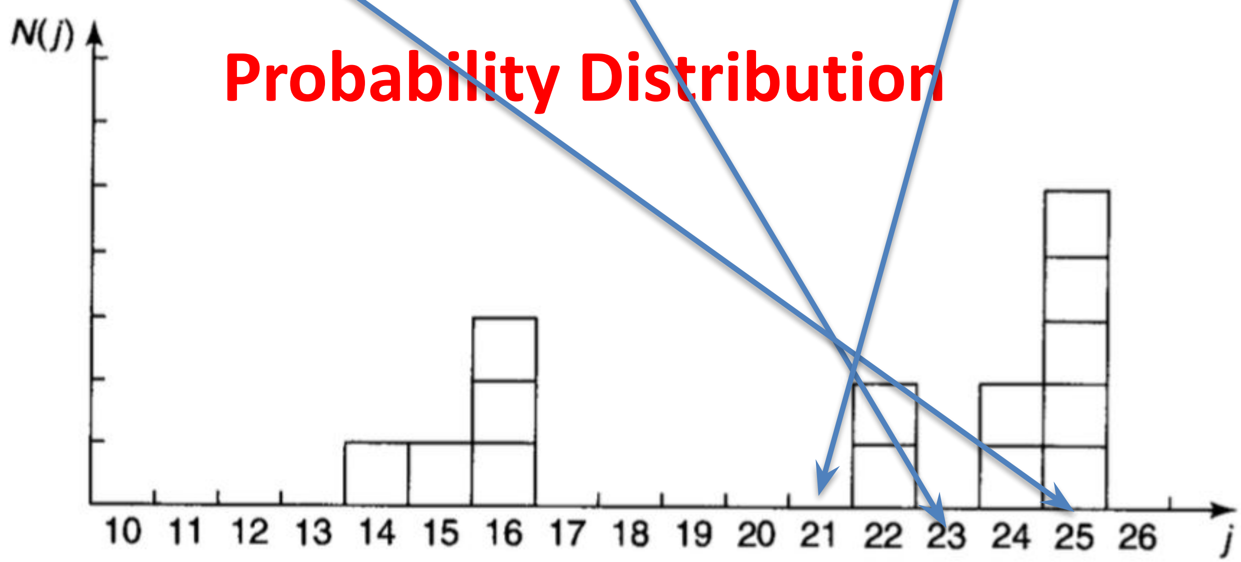
Problems in combining estimates

- We can hash multiple times, take avg. of $\boldsymbol{2^R}$ values
- Problem: ExpectedValue(2^R) $\rightarrow \infty$
  - When $\boldsymbol{2^R \geq m}$, increase $\boldsymbol{R}$ by 1 => probability halves, but value $\boldsymbol{2^R}$ doubles
    - $p = 1 - (1 - 2^{-r})^m = 1 - \ e^{-\frac{m}{2^r}}$
  - Contribution from each large $\boldsymbol{R}$ to E(2^R) grows, when R grows
  - Can have very large 2^R
- What about taking median instead?

Examples of Probability

- N people (e.g., N=14) with different age *j*
  - Probability P(j) = N(j)/N
  - Most probable? Media=? Average(mean) <*j*>=?

25.0 (14+25)/2 21.0



Combining the estimates

- Avg only: what if one very large value?
- Median: all values are power of 2
  - 1, 2, 4, 8,…,1024, 2048,…
- Solution:
  - Partition hash functions into small groups
  - Take average for each group
  - Take the median of the averages

COUNTING (MOMENTS)

Roadmap

- Motivation
- Sampling
  - Fixed-portion & fixed-size (reservoir sampling)
- Filtering
  - Bloom filter
- Counting
  - Estimating # of distinct values, moments
- Sliding window
  - Counting # of 1's in the window

Moments

- A stream S of elements drawn from a universal set: $v_1, v_2, \ldots, v_n$
  - m_i is the number of occurrences of v_i in S
  - (1,4,1,3,4,1), m_1=3, m_2 = 2;
- *k-th* moment of S:
  - $\Sigma_{i=1}^{n} (m_i)^k$

K-th moments

- 0-th moment of the stream ***S***
  - # of distinct elements in ***S***
  - **(1, 4, 1, 3, 4, 1)**
- 1^{st} moment of ***S***
  - Length of ***S***

- $\sum_{i=1}^{n}(m_i)^k$

- 2^{nd} moment of ***S*** : sum of squared frequencies
  - Surprise number, measuring the ***evenness*** of distribution of elements in ***S***

Surprise number

- Steam of 100 elements, 11 values appeared

… …
…

- Unsurprising: m_i =10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9
  - Surprise number = $10^2 + 10 * 9^2 = 910$

- $\sum_{i=1}^{n}(m_i)^k$

- Surprising: 90, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
  - Surprise number = $90^2 + 10 * 1^2 = 8{,}110$

AMS (Alon-Matias-Szegedy) Algorithm

- Estimating 2nd moment of a stream of length *n*
- Pick k random numbers between 1 to *n*
- For each, construct a variable X at position *t*
  - Record its count from position *t* onwards to *n* as X.value
- Estimate $= \mathbf{1/k \sum_{i=1}^{k} n(2x_k.value - 1)}$
  - Explain next

Example

- Stream: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b
  - Stream length n = 15
  - **a** occurs 5 times
  - **b** occurs 4 times
  - **c** occurs 3 times
  - **d** occurs 3 times
- 2^{nd} moment = $5^2 + 4^2 + 3^2 + 3^2 = 59$

Random variables

- Each random variable X records:
  - X.element: element in ***X***
  - X.value: # of occurrences of ***X*** from time ***t*** to ***n***
- X.value = 1, at time *t*
  - At time t, we have the 1^{st} encounter of this element
- X.value++, when another X.element is seen

Random variables

- Stream: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

3 8 13

- Suppose we keep three variables: X_1, X_2, X_3
  - Introduced at position: 3, 8, and 13

- Position 3: X_1.element = c, X_1.value = 1
- Position 7: X_1.value = 2

Random variables

- Stream: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

  3 8 13

- Position 8: X_2.element = d, X_2.value = 1
- Position 11: X_2.value = 2
- Position 12: X_1.value = 3
- Position 13: X_3.element = a, X_3.value = 1
- Position 14: X_3.value = 2

Random variables: final values

- X_1.value = 3, X_2.value = 2, X_3.value = 2

- Estimate of 2nd moment = ***n(2*X.value - 1)***

- Estimate using $\boldsymbol{X_1}$: 15(6-1) = 75
- Estimate using $\boldsymbol{X_2}$or $\boldsymbol{X_3}$: 15(4-1) = 45

- Avg. = (75+45+45)/3 = 55 (recall actual is 59)

Why AMS works?

- Stream: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

3 8 13 n

- ***e(i)***: element that appears in position i
  - A random variable introduced at this position will have X.element = e(i)
- ***c(i)***: # of times e(i) appears in positions $i \ldots n$
  - X.value = c(i)

- e(6) = a, c(6) = 4, // 'a' appeared 4 times in [6..n]

Why AMS works?

- Stream: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b

3 8 13

- $e(1) = a$, $c(1) = 5 = m_a$
- $e(2) = b$, $c(2) = 4 = m_b$
- $e(3) = c$, $c(3) = 3 = m_c$
- $e(4) = b$, $c(4) = m_b - 1$
- …
- $e(13) = a$, $c(13) = 2$
- $e(14) = a$, $c(14) = 1$
- $e(15) = b$, $c(15) = 1$

$1, 2, 3, \ldots, m_a$

If we can have a lot of random variables (e.g., 15)…

Why AMS works?

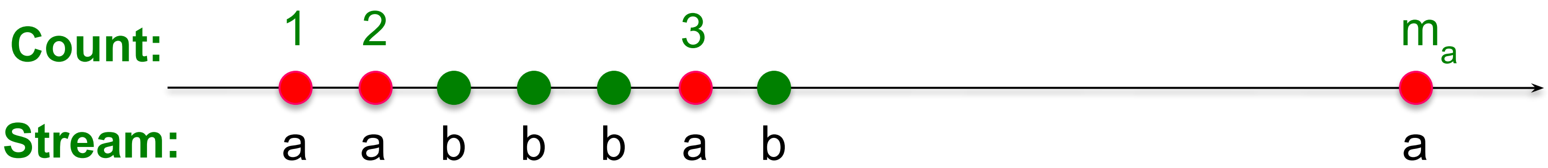
- Estimate = $\mathbf{1/k\sum_{i=1}^{k} n(2x_k.value - 1)}$
- $E\big(n(2X.value - 1)\big)$
  - average over all positions i between 1 and n

$$= \frac{1}{n}\sum_{i=1}^{n} n(2c(i) - 1)$$
$$= \sum_{i=1}^{n}(2c(i) - 1)$$
$$= \sum_{x} 1 + 3 + \cdots + (2m_x - 1)$$ for each x
$$= \sum_{x} m_x^2$$ // this is the 2^{nd} moments

- Note: 1+3+… + (2n-1) = n^2

$$\sum_{i=1}^{m_i}(2i-1) = 2\frac{m_i(m_i+1)}{2} - m_i = (m_i\ \ )^2$$

Expectation Analysis



- **2nd moment** of $S = \sum_x m_x^2$
- c_t … number of times item at time t appears from time **t** onwards ($c_1 = m_a$, $c_2 = m_a - 1$, $c_3 = m_b$)

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

- $$E[f(X)] = \frac{1}{n}\sum_{t=1}^{n} n(2c_t - 1)$$

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

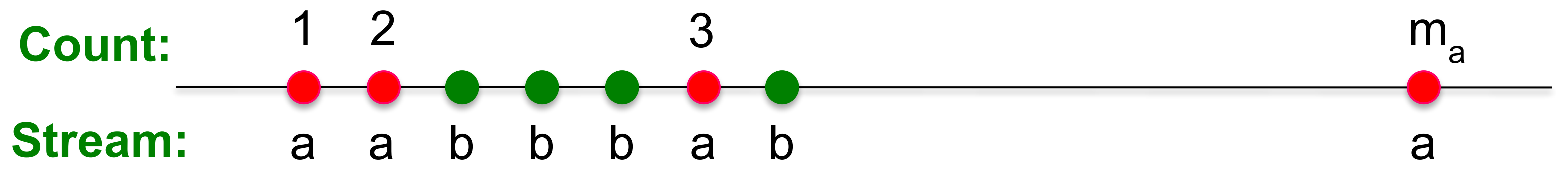
Group times by the value seen

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

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

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

Expectation Analysis



- $E[f(X)] = \frac{1}{n}\sum_x n\,(1 + 3 + 5 + \cdots + 2m_x - 1)$
  - Little side calculation: $(1 + 3 + 5 + \cdots + 2m_x - 1) = \sum_{x=1}^{m_x}(2x - 1) = 2\frac{m_x(m_x+1)}{2} - m_x = (m_x\,)^2$
- **Then $E[f(X)] = \frac{1}{n}\sum_x n\,(m_x)^2$**

- **So, $\mathbf{E}[\mathbf{f}(\mathbf{X})] = \sum_x (m_x)^2$**
- **We have the second moment (in expectation)!**

Higher-Order Moments

- **To estimate the k^{th} 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 ***m^2***
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m}(c-1)^2 = m^2$
    - **So:** $\mathbf{2c - 1 = c^2 - (c-1)^2}$
  - **For k=3:** $\mathbf{c^3 - (c-1)^3 = 3c^2 - 3c + 1}$ $\sum_{v=1}^{m} 3v^2 - 3v + 1 = m^3$
- **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 – <u>the number of inputs seen so far</u>

Streams Never End: Fixups

- **(1)** The variables ***X*** have ***n*** as a factor –
  keep ***n*** separately; just hold the count in ***X***
- **(2)** Suppose we can only store ***k*** counts.
  We must throw some ***X***s out as time goes on:
  - **Objective:**
    - Estimating 2nd moment of a stream of length ***n***
    - 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

SLIDING WINDOWS

Roadmap

- Motivation
- Sampling
  – Fixed-portion & fixed-size (reservoir sampling)
- Filtering
  – Bloom filter
- Counting
  – Estimating # of distinct values, moments
- Sliding window
  – Counting # of 1's in the window

Sliding Windows

- A useful model of stream processing is that queries are about a ***window*** of length ***N***
  - the ***N*** most recent elements received
- **Interesting case:** ***N*** is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
- **Amazon example:**
  - For every product **X** we keep 0/1 stream of whether that product was sold in the **n**-th transaction
  - We want answer queries, how many times have we sold **X** in the last **k** sales (e.g., k = 10, 20, or 200; N=100,000)

Sliding Window: 1 Stream

N = 6

- **Sliding window on a single stream:**

q w e r t y u i o p $\boxed{\text{a s d f g h}}$ j k l z x c v b n m

q w e r t y u i o p a $\boxed{\text{s d f g h j}}$ k l z x c v b n m

q w e r t y u i o p a s $\boxed{\text{d f g h j k}}$ l z x c v b n m

q w e r t y u i o p a s d $\boxed{\text{f g h j k l}}$ z x c v b n m

← Past Future →

Counting Bits (1)

- **Problem:**
  - Given a stream of **0**s and **1**s
  - Be prepared to answer queries of the form
    How many 1s are in the last *N* bits?
- **Obvious solution:**
  Store the most recent ***N*** bits
  - When new bit comes in, discard the ***N*+1**$^{\text{st}}$ bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 $\boxed{1\ 1\ 0\ 1\ 1\ 0}$ Suppose N=6

← Past Future →

Counting Bits (2)

- **You cannot get an exact answer without storing the entire window**

- **Real Problem:**
  What if we cannot afford to store *N* bits?
  - **E.g.**, we're processing 1 billion streams and ***N* = 1 billion**

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 ~~1 1 0 1 1 0~~

← Past Future →

- **But we are happy with an approximate answer**

An attempt: Simple solution

- **Q: How many 1s are in the last *N* bits?**
- A simple solution that does not really solve our problem: **Uniformity assumption**

N

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

Past Future

- **Maintain 2 counters:**
  - ***S***: number of 1s from the beginning of the stream
  - ***Z***: number of 0s from the beginning of the stream
- **How many 1s are in the last N bits?** $N \cdot \frac{S}{S+Z}$
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?
  - Cannot handle various ***k***

DGIM Method

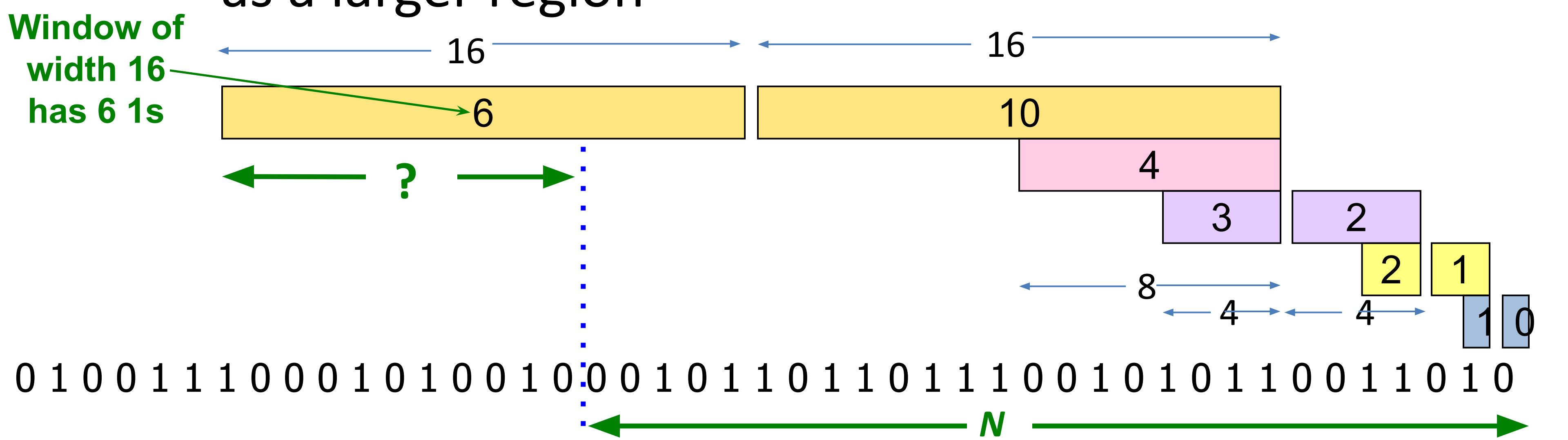
- **DGIM solution that does <u>not</u> assume uniformity**

- We store $\boldsymbol{O}(\log \boldsymbol{N} \times \log \boldsymbol{N})$ bits per stream
  - $\boldsymbol{O}(\log^2 \boldsymbol{N})$

- **Solution gives approximate answer, never off by more than 50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits

Idea: Exponential Windows

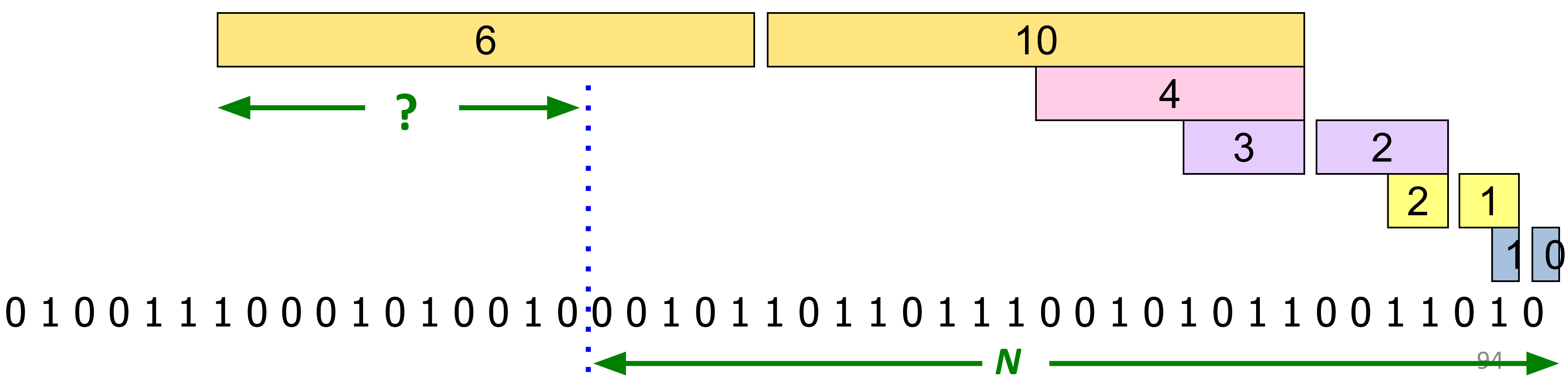
- **Solution that doesn't (quite) work:**
  - Summarize **exponentially increasing** regions of the stream, looking backward
  - Drop small regions if they begin at the same point as a larger region



We can reconstruct the count of the last ***N*** bits, except we are not sure how many of the last **6 1s** are included in the ***N***

What's Not So Good?

- As long as the **1s** are fairly evenly distributed, the error due to the unknown region is small – **no more than 50%**
- But it could be that all the **1s** are in the unknown area at the end
- In that case, **the error is unbounded!**



DGIM Algorithm

- Storage: ***O(log²N)*** bits (N is the window size)

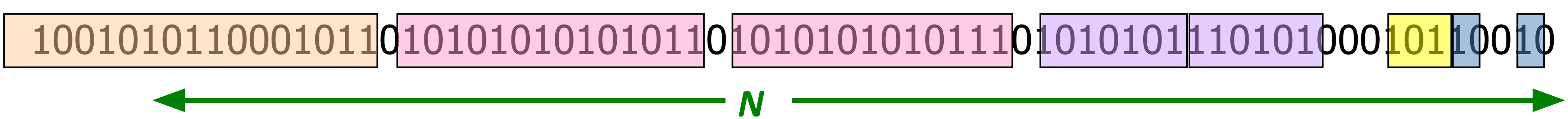
- Error rate for the queries $\leq$ **50%**

- Key idea:
  - Partition ***N*** into a (small) set of buckets
  - Remember count for each bucket
  - Use the counts to approximate answers to queries

Fixup: DGIM method

- **Idea:** Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1s**:
  - Let the block ***sizes*** (number of **1s**) increase exponentially

- **When there are few 1s in the window, block sizes stay small, so errors are small**



DGIM: Timestamps

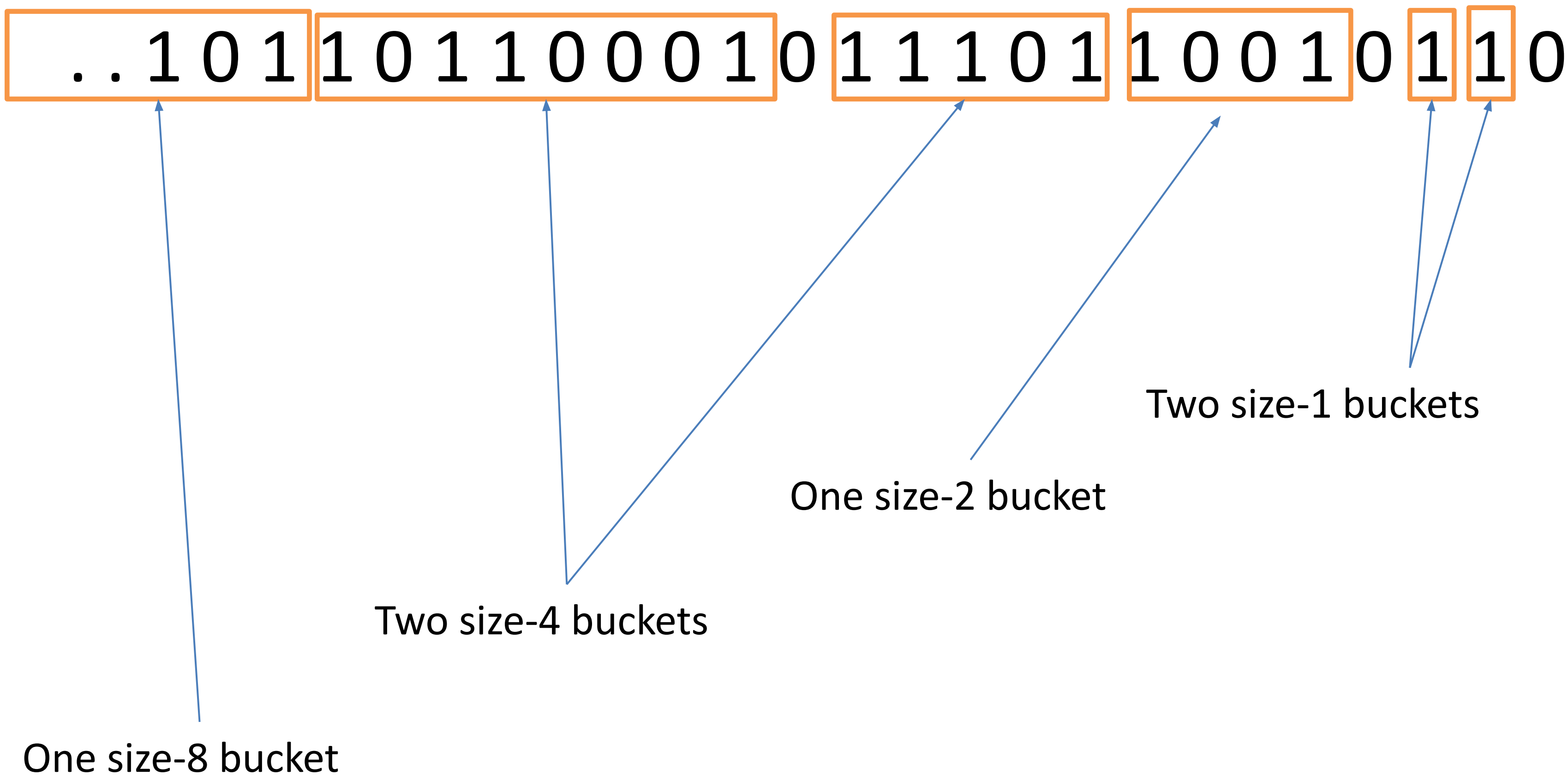
- Each bit in the stream has a ***timestamp***, starting **1**, **2,** …

- Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $\boldsymbol{O(log_2 N)}$ bits

Bucket

- Each represents a sequence of bits in window
  - It does not store the actual bits
  - Rather a timestamp and # of 1's in the sequence
- Timestamp of bucket
  - Timestamp of its end time
- Bucket size = # of 1's in the bucket
  - Always some power of 2: 1, 2, 4, …

Example Buckets



Window size N = 25

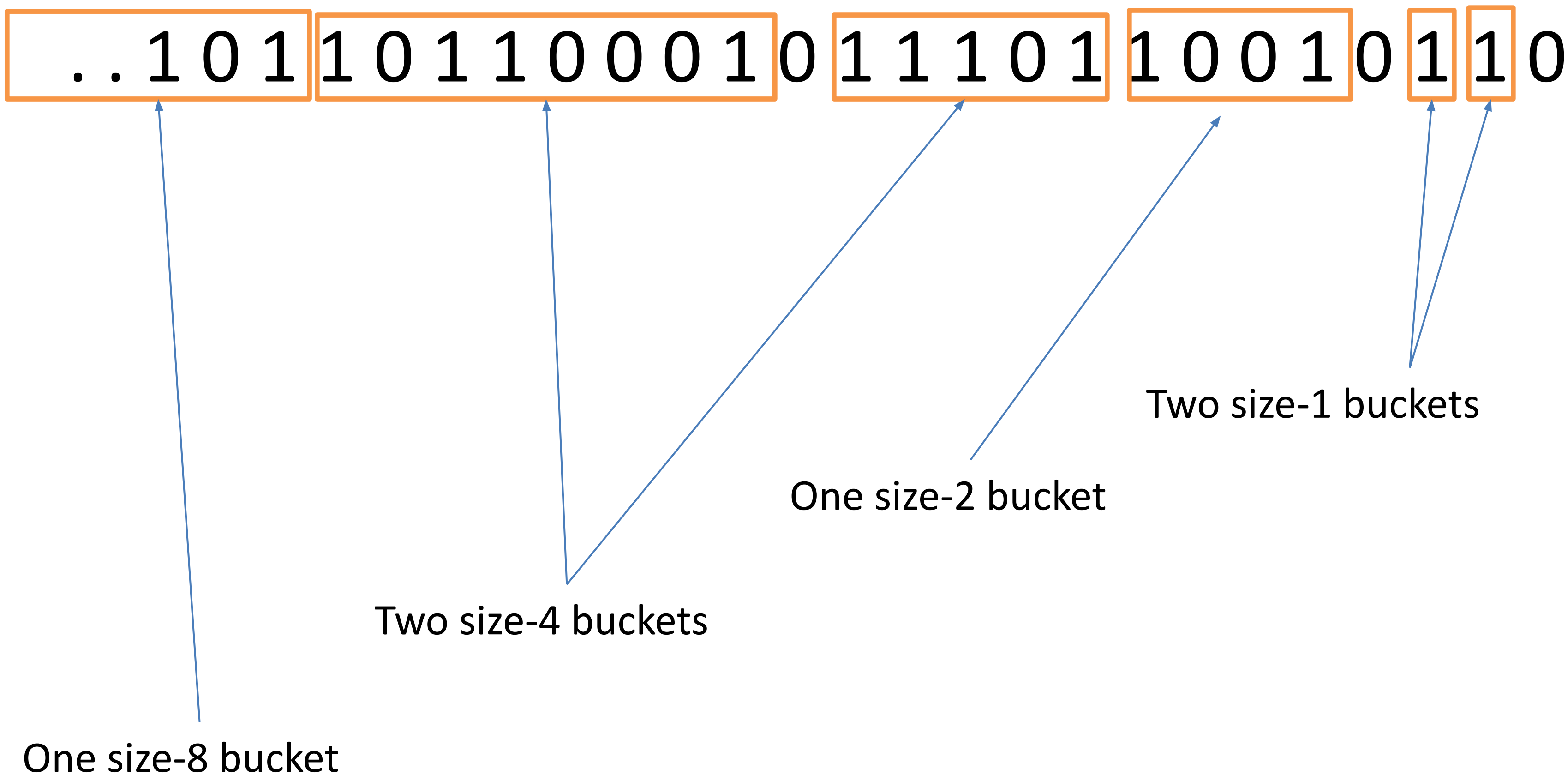
Rules for creating buckets

- Rightmost bit of each bucket = 1
- Every ***1 position*** in window is in some bucket
- A position can only be in a single bucket
- At most two buckets can have the same size
- Size of older buckets $\geq$ size of news ones

Representing a Stream by Buckets

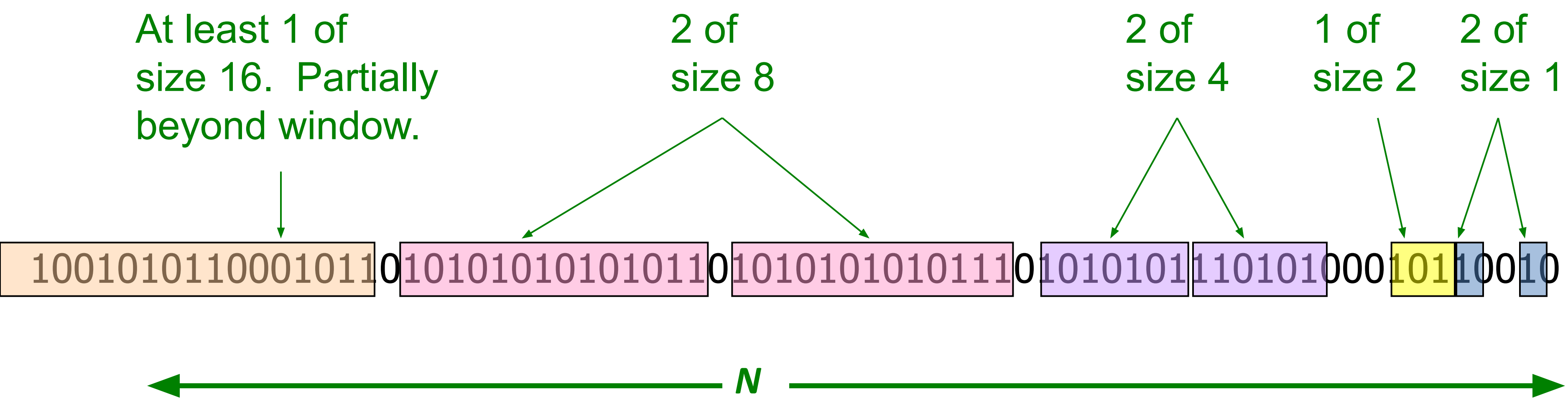
- Either **one** or **two** buckets with the same **power-of-2 number** of **1s**
- **Buckets do not overlap in timestamps**
- **Buckets are sorted by size**
  - Earlier buckets are not smaller than later buckets
- Buckets disappear when their end-time is **>** ***N*** time units in the past

Example buckets



Window size N = 25

Example: Bucketized Stream



Three properties of buckets that are maintained:

- Either **one** or **two** buckets with the same **power-of-2** number of **1s**
- Buckets do not overlap in timestamps
- Buckets are sorted by size

Updating Buckets (1)

- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to ***N*** time units before the current time

- **2 cases:** Current bit is **0** or **1**

- **If the current bit is 0:**
  no other changes are needed

Updating Buckets (2)

- **If the current bit is 1:**
  - **(1)** Create a new bucket of size **1**, for just this bit
    - **End timestamp = current time**
  - **(2)** If there are now **three buckets of size 1**, **combine the oldest two into a bucket of size 2**
  - **(3)** If there are now **three buckets of size 2**, **combine the oldest two into a bucket of size 4**
  - **(4) And so on …**

Example: Updating Buckets

Current state of the stream:

10010101100010110101010101010110101010101011101010101110101000101100010

Bit of value 1 arrives

0010101100010110101010101010110101010101011101010101110101000101100101

Two blue buckets get merged into a yellow bucket

0010101100010110101010101010110101010101011101010101110101000101100101

Next bit 1 arrives, new blue bucket is created, then 0 comes, then 1:

0101100010110101010101010110101010101011101010101110101000101100101101

Buckets get merged…

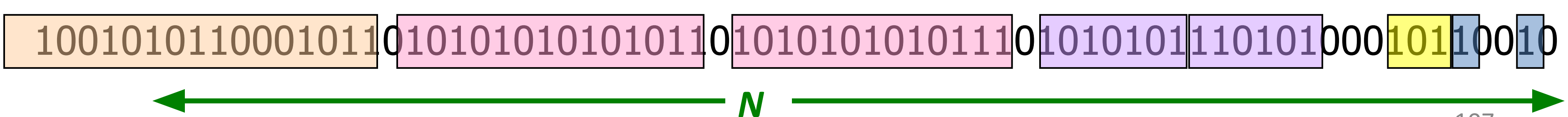
0101100010110101010101010110101010101011101010101110101000101100101101

State of the buckets after merging

0101100010110101010101010110101010101011101010101110101000101100101101

DGIM: Buckets

- A ***bucket*** in the DGIM method is a record consisting of:
  - **(A) The timestamp of its end** [O($\log_2 N$) bits]
  - **(B) The number of 1s between its beginning and end** [O($\log_2 \log_2 N$) bits]
    - $\log_2 N$ is the maximum # of bit, X, in a bucket (of size N)
    - to store X, we need $\log_2 X$ bits
- **Constraint on buckets:**
  Number of **1s** must be a power of **2**
  - That explains the **O($\log_2 \log_2 N$)** in **(B) above**



Storage for each bucket

- Timestamp: 0..N-1 (N is the window size)
  - So $\log_2 N$ bits
- Number of 1's: $2^j \le N$, $j \le \log_2 N$
  - So $\log_2(\log_2 N)$ for representing j
  - Can ignore; too small comparing to the timestamp requirement
- Each bucket requires $\approx O(\log_2 N)$

Storage requirements of DGIM

- At most 2*j buckets
  - of sizes: 2^j, 2^{j-1}, …, 1 (at most two for each size)

- Size of the largest bucket $\boldsymbol{2^j \leq N}$
  - So $\boldsymbol{j \leq log_2N}$ and $\boldsymbol{2j \leq 2log_2 N}$

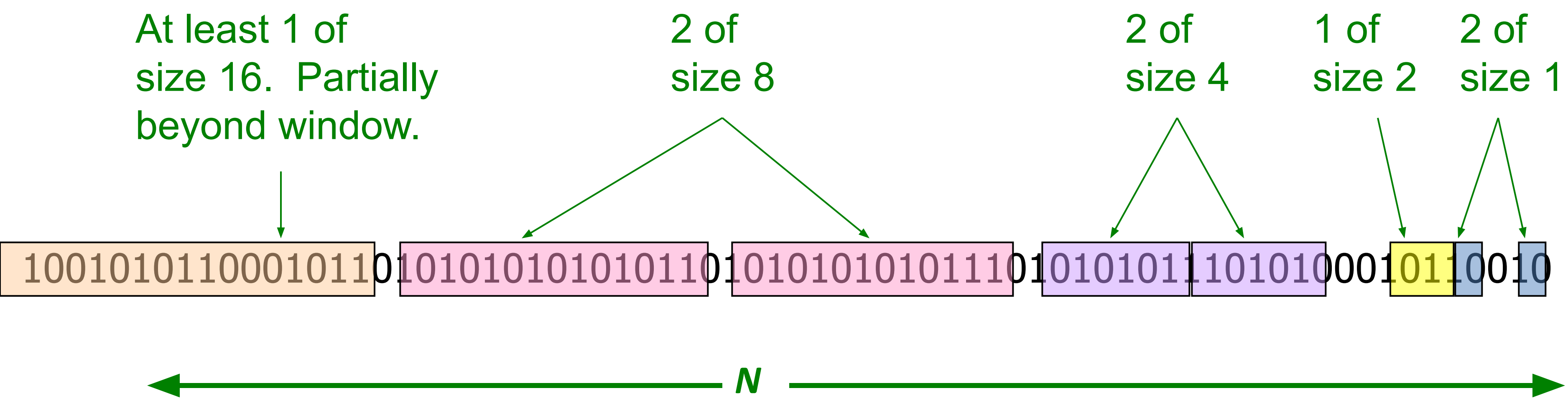
- Total storage: $\boldsymbol{O(log_2N * log_2N)}$ or $\boldsymbol{O(log^2N)}$
  - Recall that each bucket requires $\boldsymbol{O(log_2N)}$

How to Query?

- **To estimate the number of 1s in the most recent *N* bits:**
  1. **Sum the sizes of all buckets but the last**
     (note “size” means the number of 1s in the bucket)
  2. **Add half the size of the last bucket**

- **Remember:** We do not know how many **1s** of the last bucket are still within the wanted window

Example: Bucketized Stream



How close is the estimate?

- Suppose # of 1's in the last bucket ***b*** = $\boldsymbol{2^j}$
- ***Case 1: estimate < actual value c***
  - Worst case: all 1's in bucket ***b*** are within range
  - So estimate missed "at most half of $\boldsymbol{2^j}$" -> $\boldsymbol{2^{j-1}}$
  - ***Want to show c $\geq$ 2^j , so what's the minimum C?***
    - C has at least one 1 from ***b***, plus at least one of buckets of lower powers: $2^{j-1} + 2^{j-2} \ldots + 1 = 2^j - 1$; $c >= 1 + 2^j - 1$; ***missed at most*** 2^{j-1}
    - $1 + 2 + 4 + .. + 2^{r-1} = 2^r - 1$
  - So estimate missed ***at most 50% of c***
  - That is, ***the estimate is at least 50% of c***

How close is the estimate?

- Suppose # of 1's in the last bucket $\boldsymbol{b = 2^j}$

- ***Case 2: estimate > actual value c***
  - Worst case: only rightmost bit of b is within range
  - And only one bucket for each smaller power
  - $\boldsymbol{c} = 1 + 2^{j-1} + 2^{j-2} + \ldots + 1 = 1 + 2^j - 1 = 2^j$
  - Estimate = 2^{j-1} (last bucket) + $2^{j-1} + \ldots + 1$

  = $\boldsymbol{2^j}$ ***– 1 (c minus the right most bit) +*** $\boldsymbol{2^{j-1}}$ (last bucket)
  - $\boldsymbol{2^{j-1} + 2^{j-2} \ldots + 1 = 2^j - 1}$
  - So ***estimate is no more than 50% greater than c***

Extensions

- Can we use the same trick to answer queries **How many 1's in the last *k*?** where ***k* < *N***?
  - **A:** Find earliest bucket **B** that at overlaps with ***k***. Number of **1s** is the **sum of sizes of more recent buckets + ½ size of B**