CMSC5741 Big Data Tech. & Apps.

Lecture 4: Mining Data Streams

Prof. Michael R. Lyu

Computer Science & Engineering Dept.

The Chinese University of Hong Kong

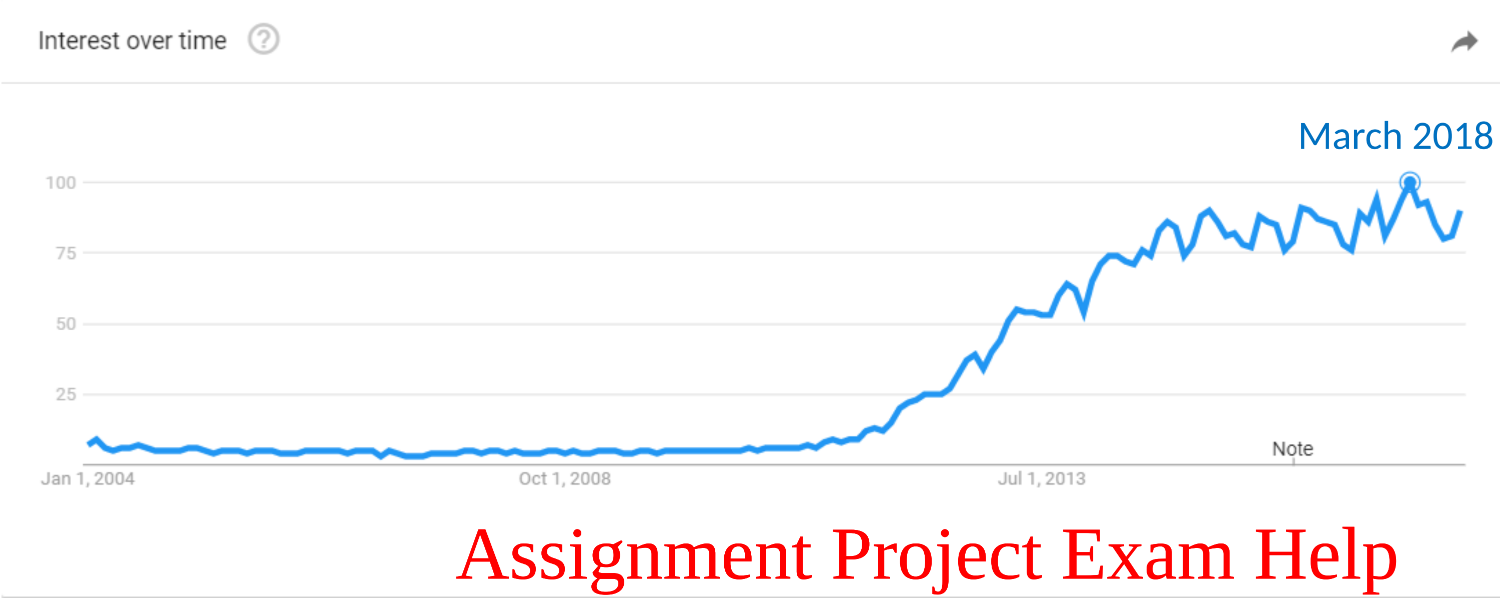
Motivation

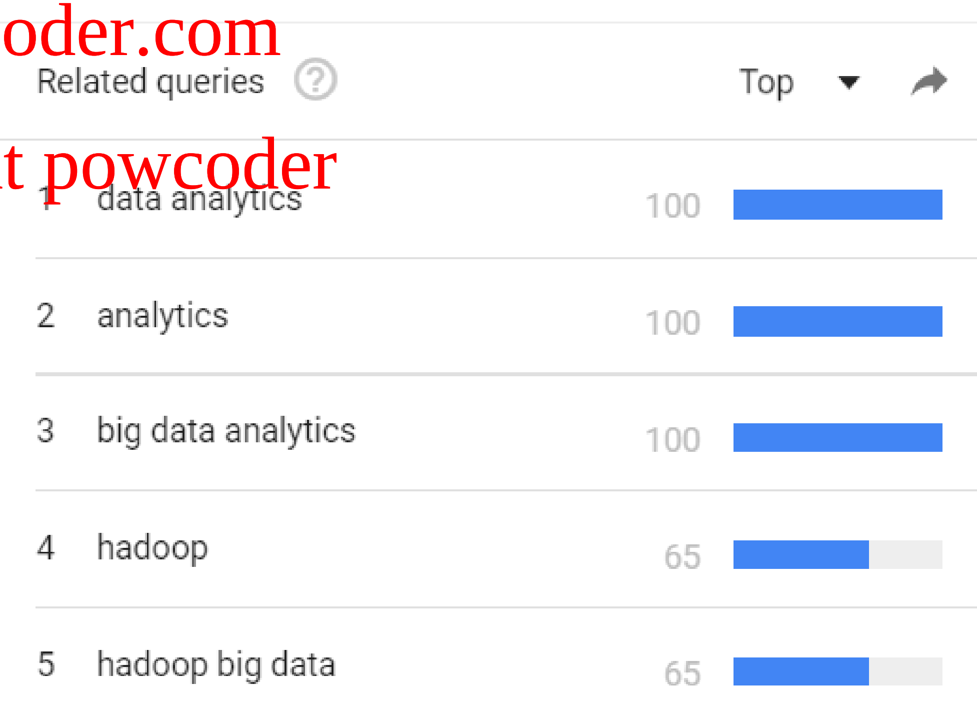
- In many data mining situations, we know the entire data set in advance

- Stream Management is important when the input rate is controlled externally:
  - Google queries
  - Twitter and Facebook status updates
- We can think of the data as infinite and non-stationary (the distribution changes over time)

Google Trends

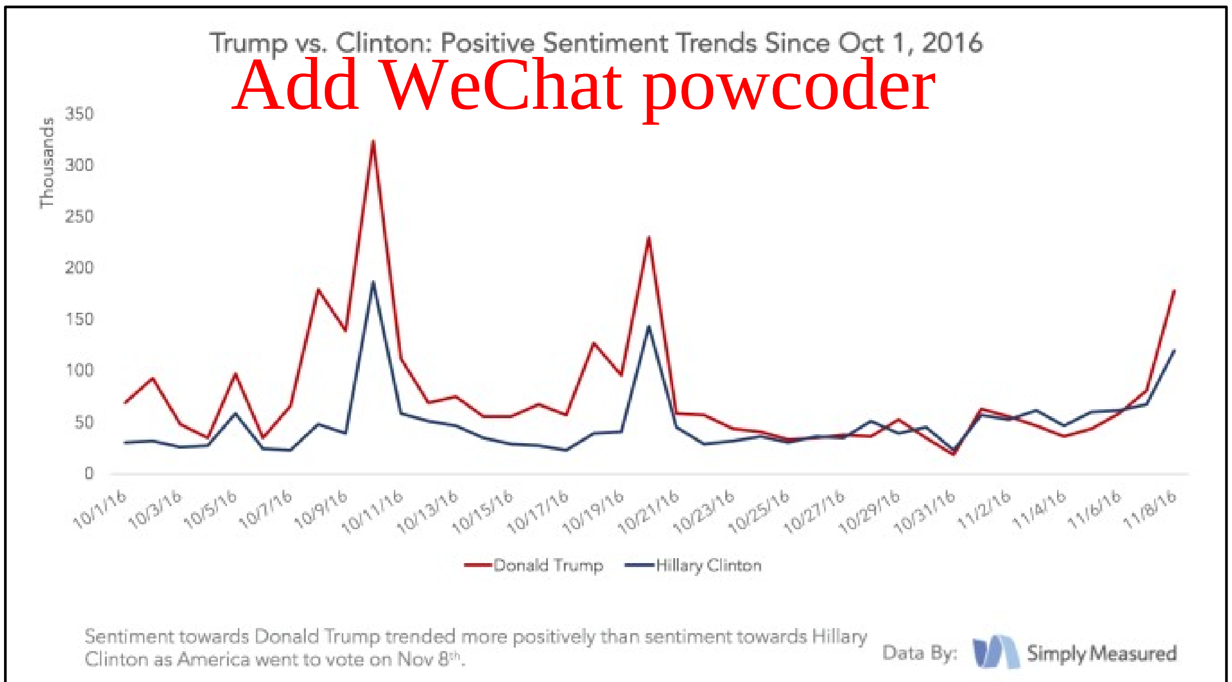
When we search for “big data”





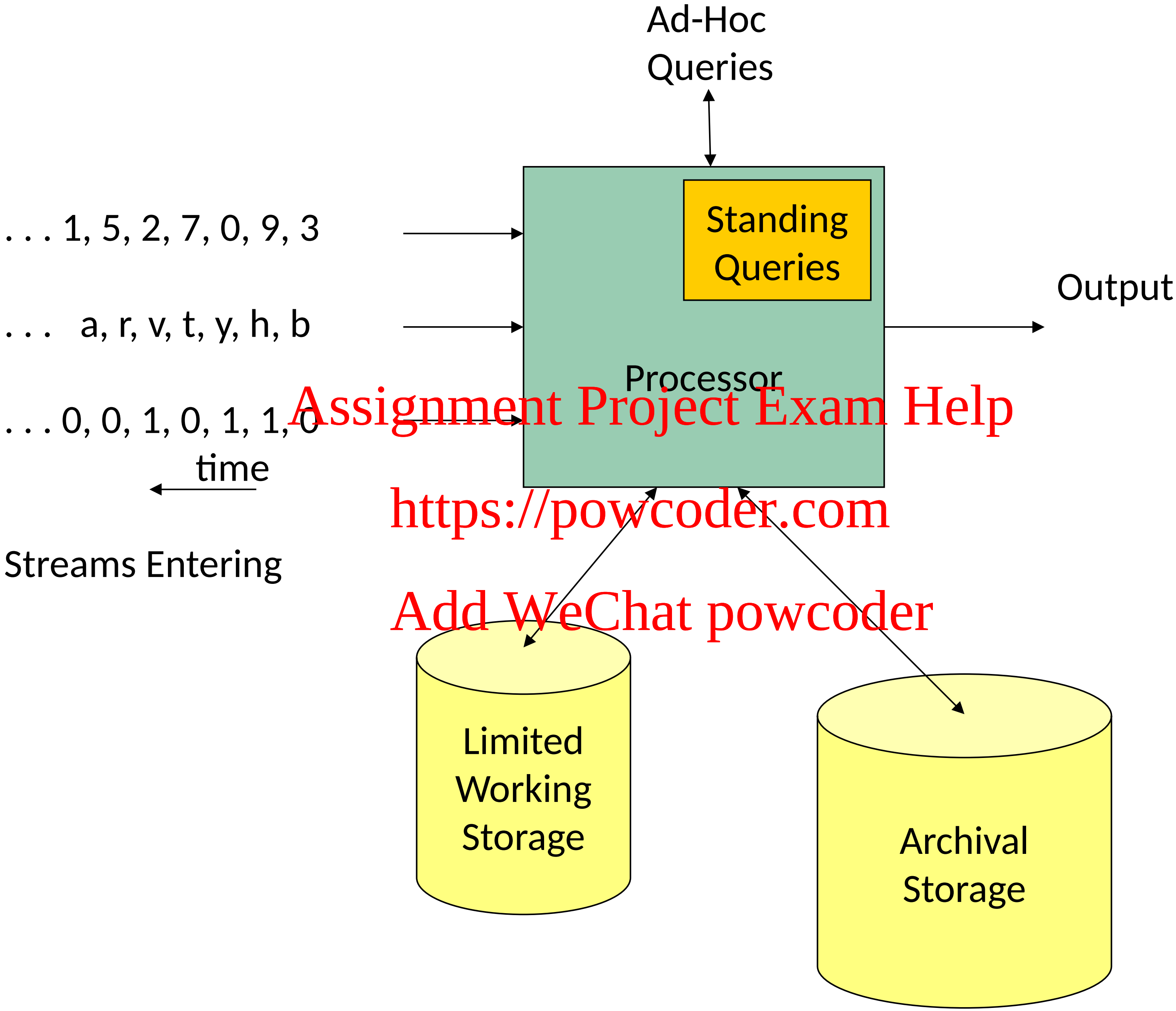
Election 2016: Trump vs Clinton


Trump vs. Clinton: Negative Sentiment Trends Since Oct 1, 2016
Thousands
80
70
60
50
40
30
20
10
0
10/1/16
10/3/16
10/5/16
10/7/16
10/9/16
10/11/16
10/13/16
10/15/16
10/17/16
10/19/16
10/21/16
10/23/16
10/25/16
10/27/16
10/29/16
10/31/16
11/2/16
11/4/16
11/6/16
11/8/16
Donald Trump
Hillary Clinton
Donald Trump trended more negatively than Hillary Clinton until the final debate.
However, negative sentiment was close for the candidates going into Election Day.
Data By: Simply Measured


Trump vs. Clinton: Positive Sentiment Trends Since Oct 1, 2016
Thousands
350
300
250
200
150
100
50
0
10/1/16
10/3/16
10/5/16
10/7/16
10/9/16
10/11/16
10/13/16
10/15/16
10/17/16
10/19/16
10/21/16
10/23/16
10/25/16
10/27/16
10/29/16
10/31/16
11/2/16
11/4/16
11/6/16
11/8/16
Donald Trump
Hillary Clinton
Sentiment towards Donald Trump trended more positively than sentiment towards Hillary Clinton as America went to vote on Nov 8th.
Data By: Simply Measured

The Stream Model

- Input tuples (e.g., [user, query, time]) enter at a rapid rate, at one or more input ports
- The system cannot store the entire stream accessibly
- How do you make critical calculations about the stream using a limited amount of (secondary) memory?


Ad-Hoc
Queries
Standing
Queries
. . . 1, 5, 2, 7, 0, 9, 3
. . .   a, r, v, t, y, h, b
. . . 0, 0, 1, 0, 1, 1, 0
time
Streams Entering
Processor
Output
Limited
Working
Storage
Archival
Storage

Problems on Data Streams

- Types of queries one wants on answer on a stream:

  - 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

Problems on Data Streams

- Types of queries one wants on answer on a 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
  - Finding frequent elements

Applications (1)

- Mining query streams
  - Google wants to know what queries are more frequent today than yesterday
- Mining click streams
  - Yahoo! wants to know which of its pages are getting an unusual number of hits in the past hour
- Mining social network news feeds
  - E.g., look for trending topics on Twitter, Facebook

Applications (2)

- Sensors Networks
  - Many sensors feeding into a central controller
- Telephone call records
  - Data feeds into customer bills as well as settlements between telephone companies
- IP packets can be monitored at a switch
  - Gather information for optimal routing
  - Detect denial-of-service attacks

Outline

- Sampling from a Data Stream
- Queries over a (long) Sliding Windows
- Filtering Data Streams
- Counting Distinct Elements
- Computing Moments
- Counting Itemsets

Outline

- Sampling from a Data Stream
- Queries over a (long) Sliding Windows
- Filtering Data Streams
- Counting Distinct Elements
- Computing Moments
- Counting Itemsets

Sampling from a Data Stream

- Since we cannot store the entire stream, one obvious approach is to store a sample
- Two different problems:
  - Sample a fixed proportion of elements in the stream (say 1 in 10)
  - Maintain a random sample of fixed size over a potentially infinite stream
    - At any "time" *t* we would like a random sample of *n* elements. For all *t*, each of *n* elements seen so far has equal prob. of being sampled

Sampling a Fixed Proportion

- Problem 1: Sampling fixed proportion
- Scenario: Search engine query stream
  - Stream of tuples: (user, query, time)
  - Answer questions such as: How often did a user run the same query on two different days?
  - Have space to store 1/10th of query stream
- Naive solution:
  - Generate a random integer in [0..9] for each query
  - Store the query if the integer is 0, otherwise discard

Problem with Naive Approach

- Simple question: What fraction of queries by an average user are duplicates?
- Suppose each user issues s queries once and d queries twice (total of $s+2d$ queries), sample rate is p
  - Correct answer: $d/(s+d)$
  - Sample will contain sp of the singleton queries and $2dp$ of the duplicate queries at least once
  - But only dp^2 pairs of duplicates
    - $dp^2 = p * p * d$
  - Of d "duplicates" $2p(1-p)d$ appear once
    - $2p(1-p)d = ((p*(1-p))+((1-p)*p))*d$
  - So the sample-based answer is: $dp^2/(sp+dp^2+ 2p(1-p)d)$

Problem with Naive Approach

- A concrete example:
  - Query stream: 1, 2, 3, 4, 5, 6, 7, 7, 8, 8
  - Sample 50% of the queries in this case
  - Correct answer: 2/(6+2) = 25% are duplicates
  - If our sample is 1, 2, 3, 4, 5, then 0% are duplicates
  - If our sample is 6, 7, 7, 8, 8, then 67% are duplicates
  - What is the expectation of fraction of duplicates if we use sample-based method?

Answer: 1/9

Solution?

Solution: Sample Users

- Pick 1/10th of users and take all their searches in the sample

- Use a hash function that hashed the user name or user id uniformly into 10 buckets

- Generalized: Pick 1/dth of users, we need to use d buckets

Generalized Solution

- Stream of tuples with keys:
  - Key is some subset of each tuple's components
    - E.g., tuple is (user, search, time); key is user
  - Choice of key depends on application
- To get a sample of size a/b ($a<b$):
  - Hash each tuple's key uniformly into b buckets
  - Pick the tuple if its hash value is at most a

    ($h(x) = 1, 2, \ldots, a$)

Maintaining a Fixed-size Sample

- Problem 2: Fixed-size sample
- Suppose we need to maintain a sample S of size exactly s (s is fixed; e.g., s=10 items out of S=100 space)
  - – E.g., main memory size constraint
- Why? Don't know length of stream in advance
  - – In fact, stream could be infinite
- Suppose at time t we have seen n items
  - – Ensure each item is in the sample S with equal prob. s/n

Solution: Fixed Size Sample

- Algorithm:
  - Store all the first s elements of the stream to S

  - Suppose we have seen *n* elements, and now the $n+1^{th}$ element arrives (n s)

    - With prob. $s/n+1$, pick the $n+1^{th}$ element, else discard it

    - If we picked the $n+1^{th}$ 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, i.e., each item is in the sample *S* with equal prob.

Proof: By Induction

- We prove this by induction:
  - Assume that after n elements, the sample contains each element seen so far with prob. 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 prob. $s/(n+1)$
  - Obviously, after we see $n=s$ elements the sample has the wanted property
    - Each out of n=s elements is in the sample with prob. $s/s=1$

Proof: By Induction

- After n elements, the sample S contains each element seen so far with probability s/n
- Now element n+1 arrives
- For elements already in S, probability of remaining in S is:

- 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 =

Outline

- Sampling from a Data Stream
- Queries over a (long) Sliding Windows
- Filtering Data Streams
- Counting Distinct Elements
- Computing Moments
- Counting Itemsets

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 it cannot be stored in memory, or even on disk

  - Or, there are so many streams that windows for all cannot be stored

A Sliding Window Example

N = 6

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

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 are processing 1 billion streams and N = 1 billion

- But we're happy with an approximate answer

An Attempt: Simple Solution

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

- Maintain 2 counters:
  - *S*: number of 1s so far

  - *Z*: number of 0s so far
- How many 1s are in the last *N* bits? $N \cdot S/(S+Z)$
- But, what if stream is non-uniform?
  - What if distribution changes over time?

DGIM Method

- Store O($\log^2 N$) bits per stream

- Gives approximate answer, never off by more than 50%

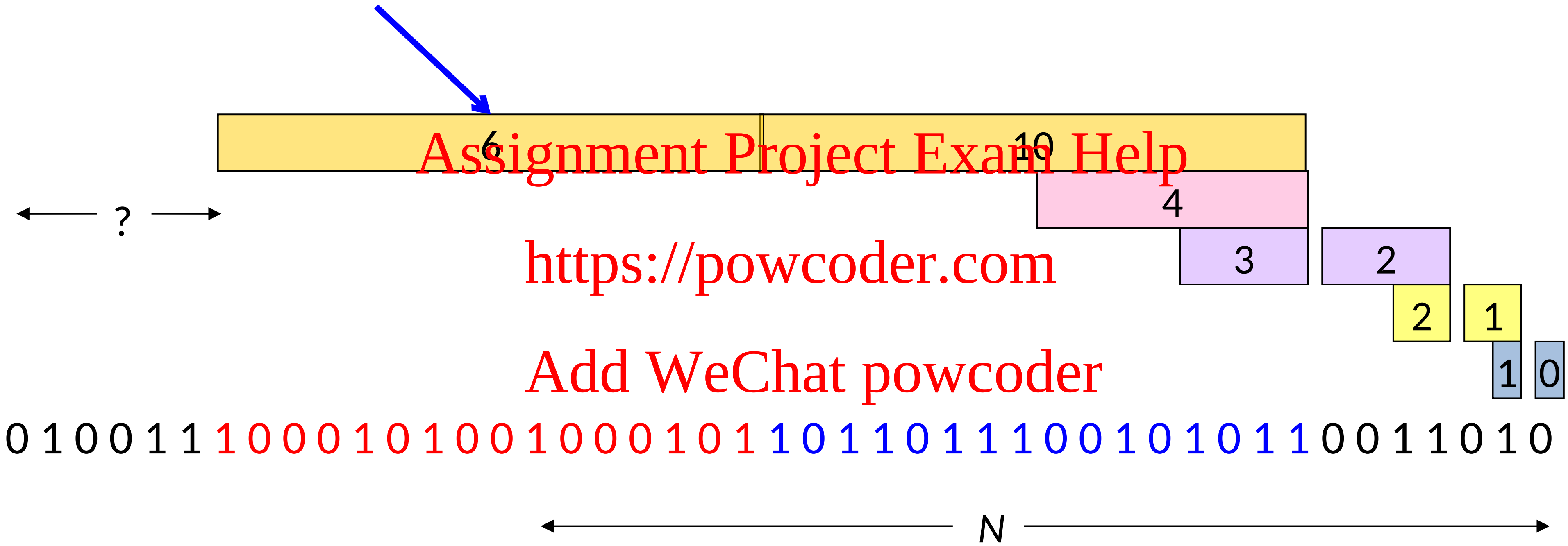
  – Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits

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

An Exponential Window Example

Window of width 16 has 6 1s



We can construct the count of the last *N* bits, except we're not sure how many of the last 6 are included.

What's Good?

- Stores only O($\log^2 N$) bits
  - O($\log N$) counts of $\log N$ bits each

- Easy update as more bits enter

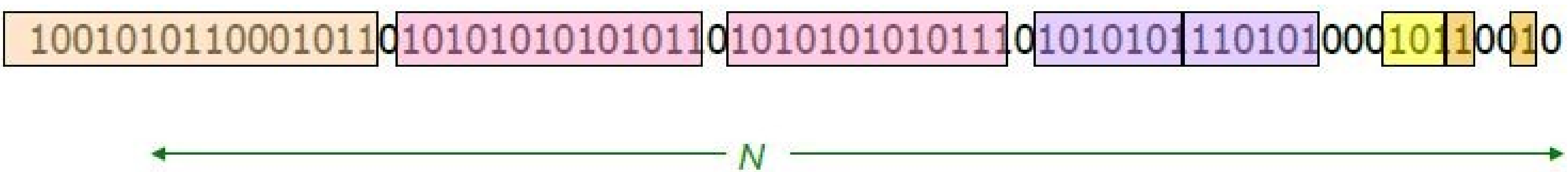
- Error in count no greater than the number of 1s in the "unknown" area

What's Not So Good?

- As long as the 1s are fairly evenly distributed, the error ratio 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

Fixup: DGIM Method

- Instead of summarizing fixed-length blocks, summarize blocks with specific numbers 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 $O(\log_2 N)$ bits

DGIM: Buckets

- A *bucket* in the DGIM method is a record consisting of:

  1. The timestamp of its end [O(log *N*) bits]

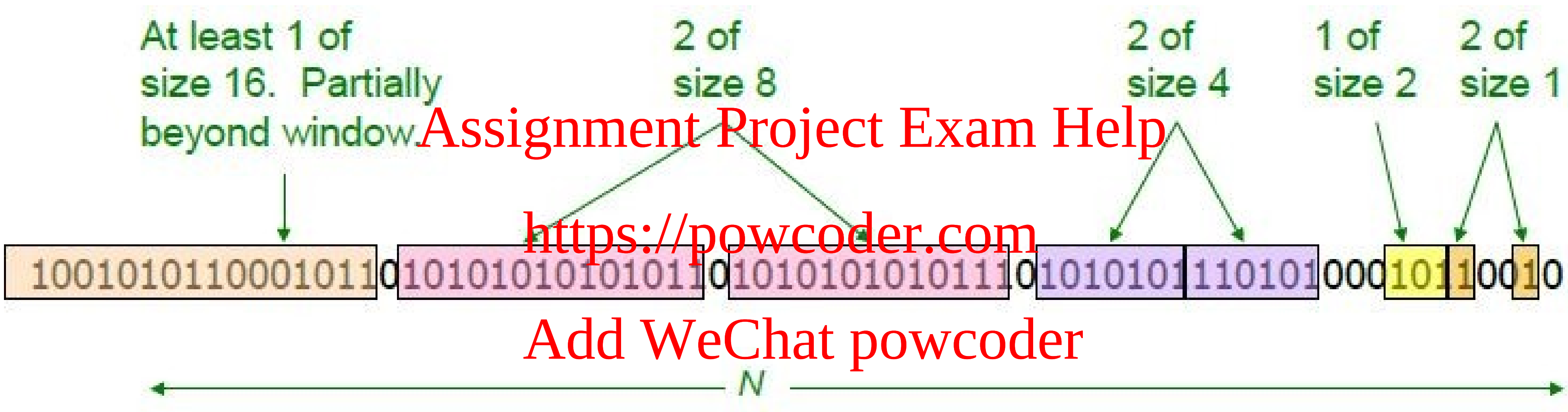
  2. The number of 1's between its beginning and end: [O(log log *N*) bits]

- Constraint on buckets: Number of 1s must be a power of 2
  - That explains the O(log log *N*) in (2)

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: Bucketized Stream



Properties we maintain:

- Either one or two buckets with the same power-of-2 number of 1s
- Buckets do not overlap in timestamp
- 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

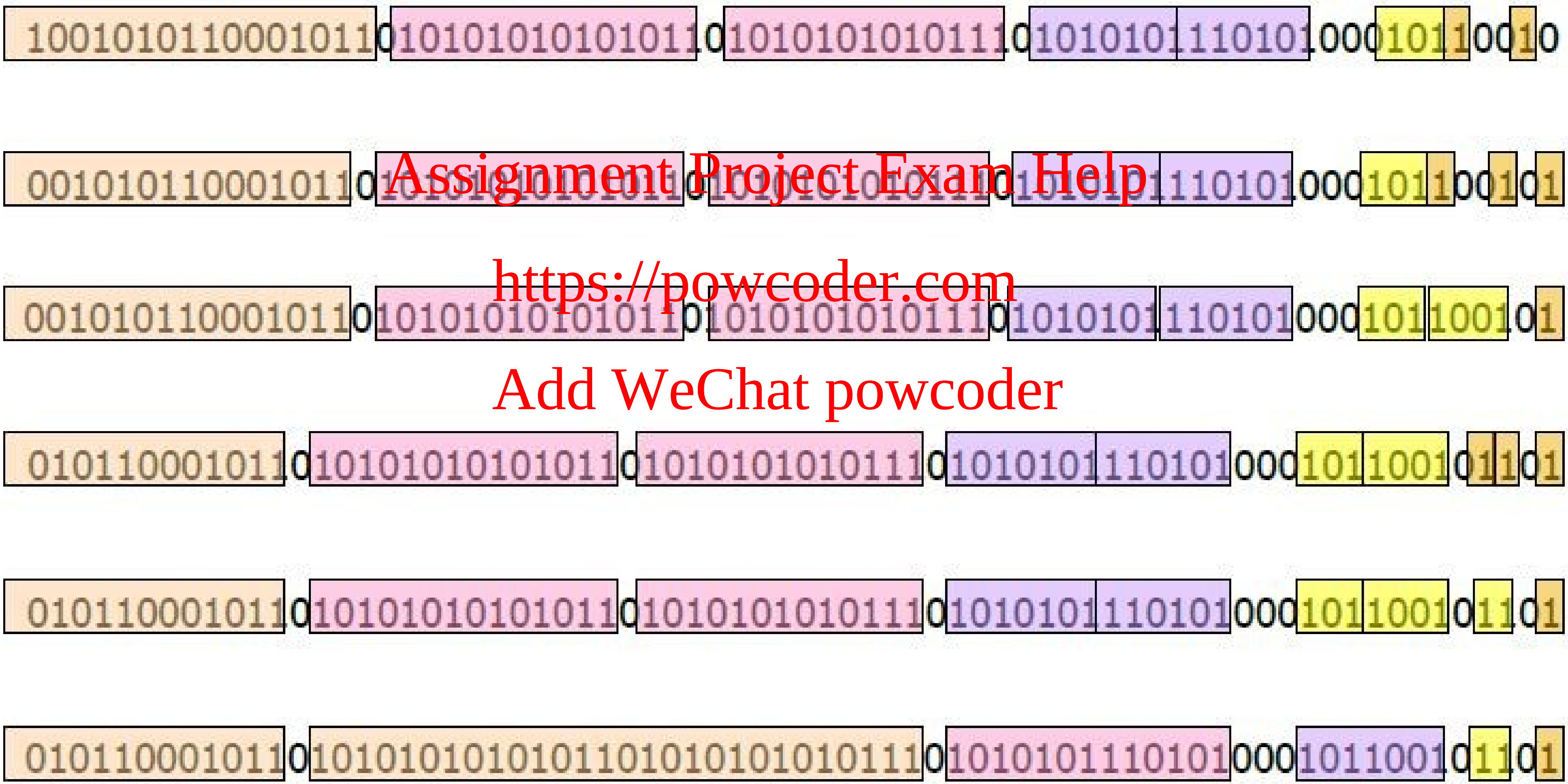
- 2 cases: Current bit is 0 or 1

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

Updating Buckets – (2)

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

Example



10010101100010110101010101010101101010101010101110101010111010100010111001 0
0010101100010110101010101010101101010101010101110101010111010100010110010 1 0 1
00101011000101101010101010101011010101010101011101010101110101000101100101 0 1
010110001011010101010101010110101010101010111010101011101010001011001 0 1 1 0 1
0101100010110101010101010101101010101010101110101010111010100010110010110 1
01011000101101010101010101011010101010101011101010101110101000101100101101

How to Query?

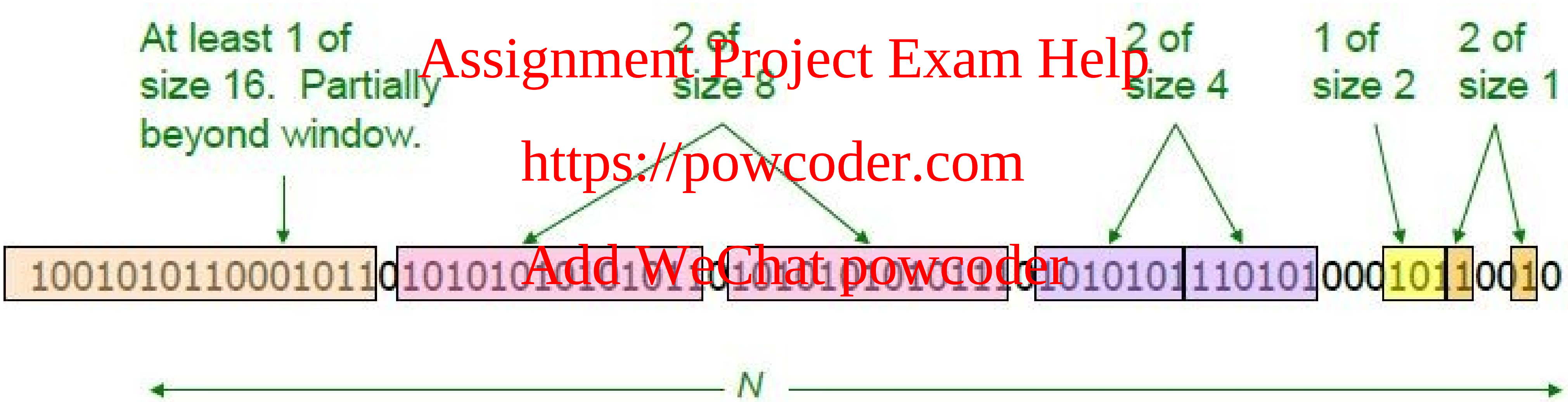
- To estimate the number of 1s in the most recent N bits:

  - Sum the sizes of all buckets but the last

  - Add half the size of the last bucket

- Remember: we don't know how many 1s of the last bucket are still within the window

Example: Bucketized Stream



In-Class Practice 1

- Go to practice

Error Bound: Proof

- Suppose the last bucket has size 2^r
- Then by assuming 2^{r-1} of its 1s are still within the window, we make an error of at most 2^{r-1}
- Since there is at least one bucket of each of the sizes less than 2^r, the true sum is no less than $1 + 2 + 4 + \ldots + 2^{r-1} = 2^r - 1$
- Thus, error ratio is at most $2^{r-1} / (2^r - 1) \approx 50\%$

Extensions (For Thinking)

- Can we use the same trick to answer queries “How many 1s in the last k?” where $k < N$?

- Can we handle the case where the stream is not bits, but integers, and we want the sum of the last k?

Reducing the Error

- Instead of maintaining 1 or 2 of each size bucket, we allow either r-1 or r for r > 2
  - Except for the largest size buckets; we can have any number between 1 and r of those
- Error is at most 1/r
- By picking r appropriately, we can tradeoff between number of bits and the error

Outline

- Sampling from a Data Stream
- Queries over a (long) Sliding Windows
- **Filtering Data Streams**
- Counting Distinct Elements
- Computing Moments
- Counting Itemsets

Filtering Data Streams

- Each element of data stream is a tuple (a finite list of elements)
- Given a list of keys *S*

- How to determine which elements of stream have keys 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

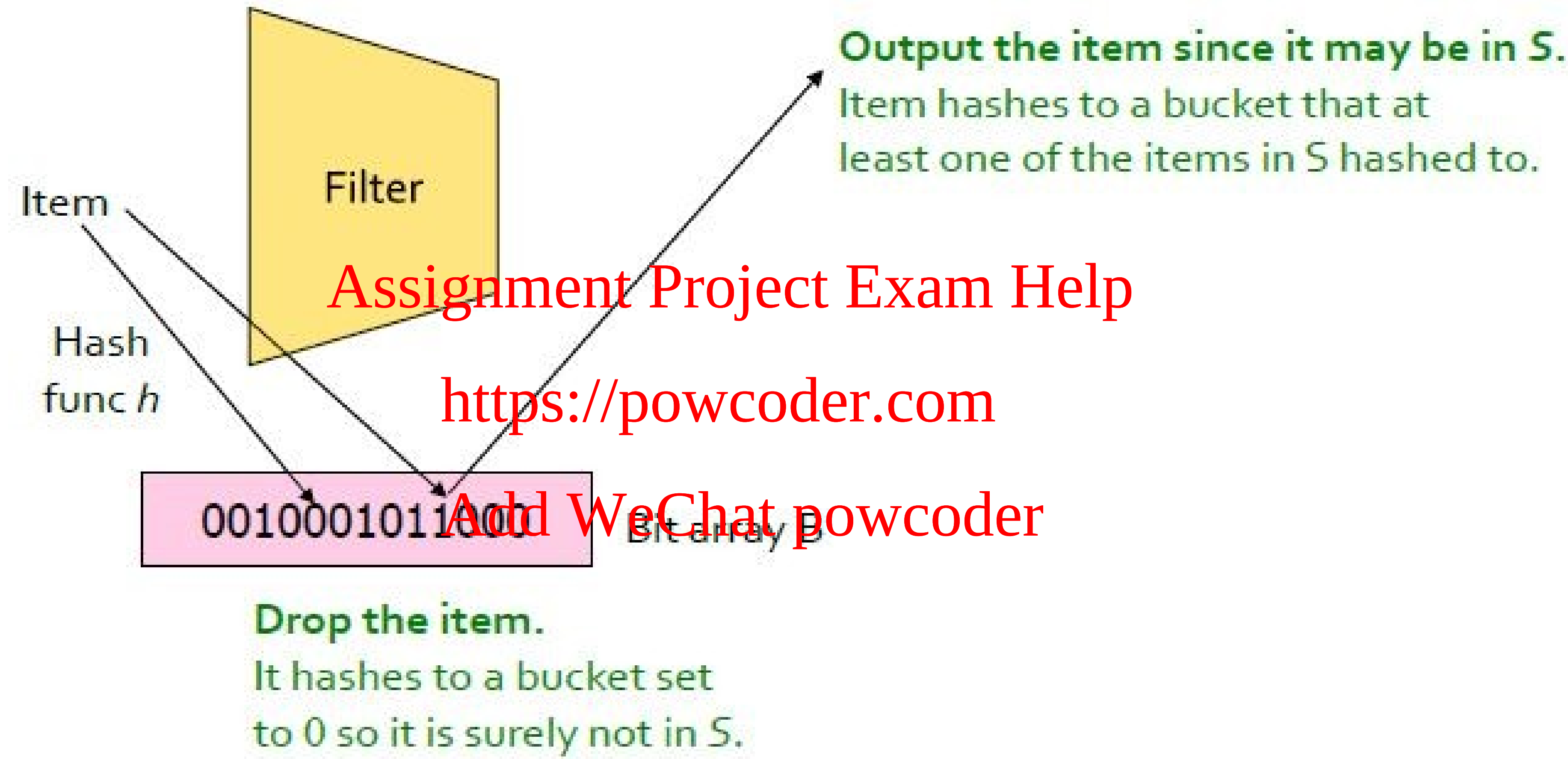
Applications

- 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
  - 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 filter
- Create a bit array *B* of *n* bits, initially all 0s
- Choose a hash function h with range *[0,n)*
- Hash each member of *s* *S* to one of m 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 bucked that has the bit set to 1, so it always gets through (no false negatives)

- 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)
  - Actually, less than 1/8th, because more than one address might hash to the same bit

Analysis: Throwing Darts

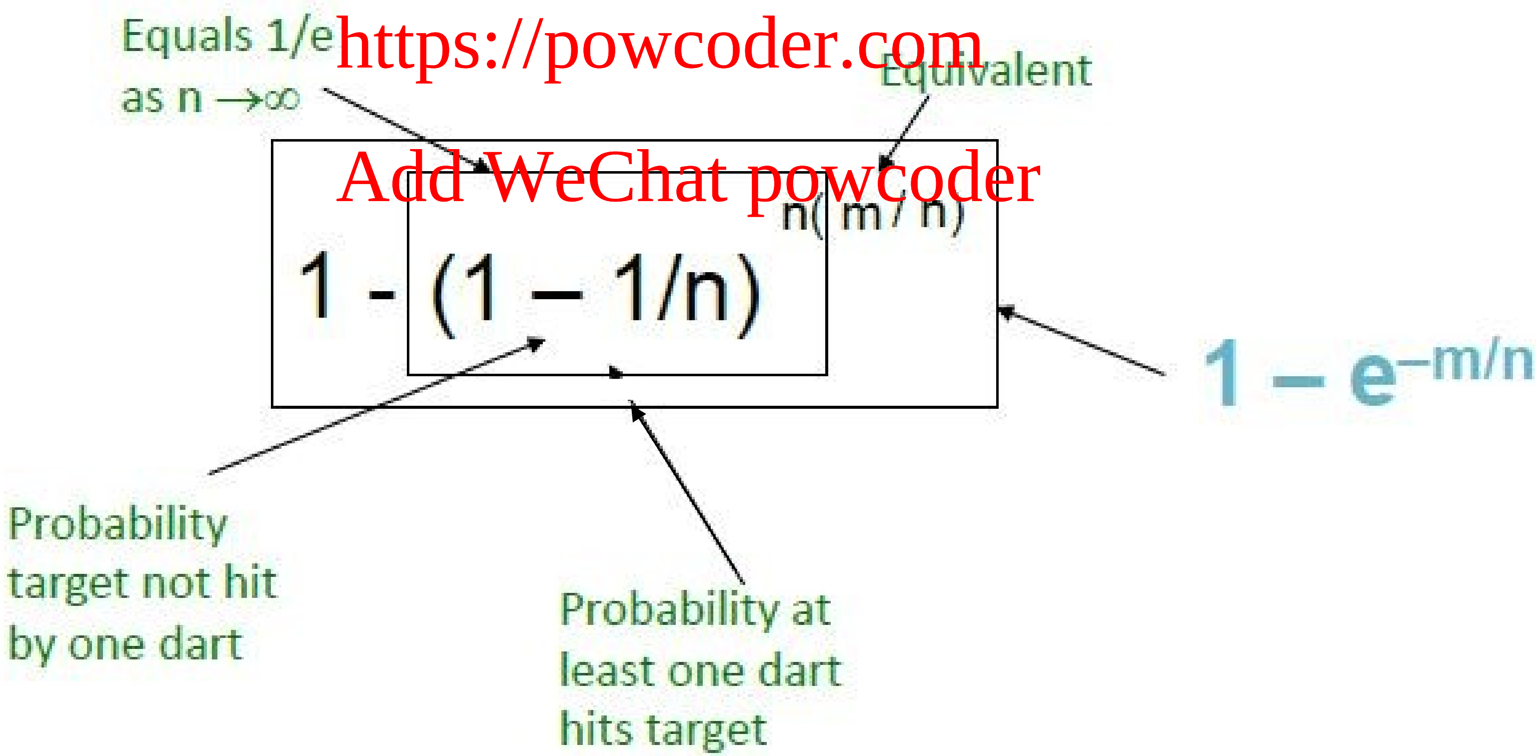
- More accurate analysis for the number of false positives

- 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?



Analysis: Throwing Darts – (3)

- Fraction of 1s in the array *B* == probability of false positive ==

- Example: darts, targets

  – Fraction of 1s in B = = 0.1175
  – Compare with our earlier estimate: 1/8 = 0.125

- Can we improve this error?

Bloom Filter

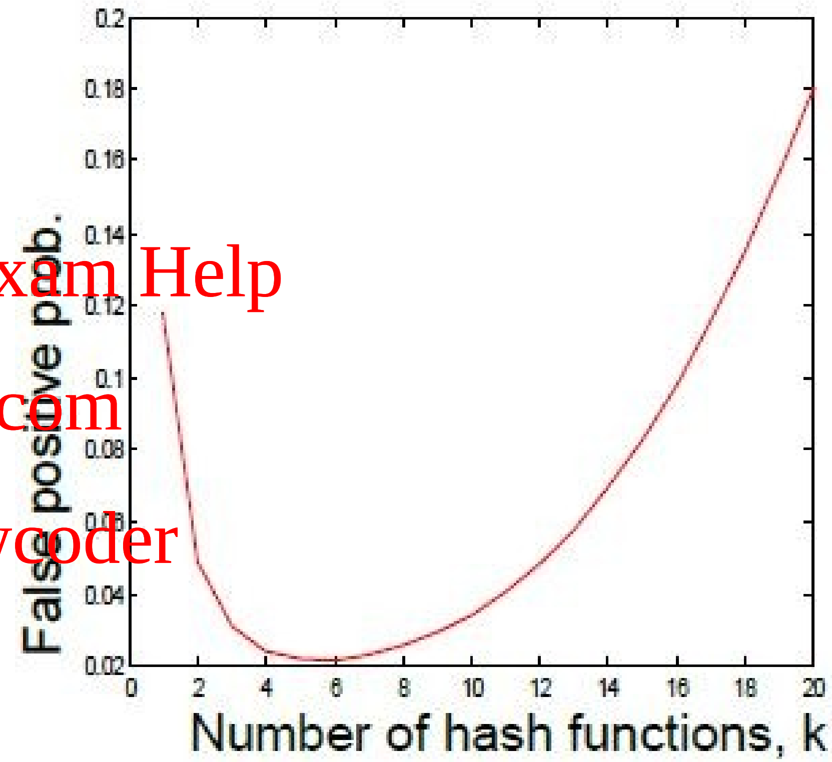
- Consider: $|S| = m$, $|B| = n$
- Use k independent hash functions
- Initialization:
  - Set B to all 0s
  - Hash each element using each hash function , set (for each)
- Run-time:
  - When a stream element with key x arrives
    - If for all , then declare that x is in S
    - Otherwise discard the 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 1s is
- But we have k independent hash functions
- So, false positive probability =

Bloom Filter – Analysis (2)

- m = 1 billion, n = 8 billion
  - k = 1: = 0.1175
  - k = 2: = 0.0493



- What happens as we keep increasing k?
- “Optimal” value of k:
  - E.g.:

Bloom Filter: Wrap-up

- Bloom filters guarantee no false negatives, and use limited memory
  - Great for pre-processing before more expensive checks
  - E.g., Google's BigTable, Squid web proxy
- Suitable for hardware implementation
  - Hash function computations can be parallelized

Outline

- Sampling from a Data Stream
- Queries over a (long) Sliding Windows
- Filtering Data Streams
- Counting Distinct Elements
- Computing Moments
- Counting Itemsets

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
- Obvious approach:
  - Maintain the set of elements seen so far

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?

Using Small Storage

- Real Problem: What if we do not have space to store the complete set?

- Estimate the count in an unbiased way

- Accept that the count may be in error, but limit the probability that the error is large

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
- Record R = the maximum $r(a)$ seen
  – $R = \max_a r(a)$, over all the items a seen so far
- Estimated number of distinct elements = 2^R

Based on a variant due to AMS (Alon, Matias, and Szegedy)

Why It Works

- The probability that a given $h(a)$ ends in at least r 0s is 2^{-r}
  - $h(a)$ hashes elements uniformly at random
  - Probability that a random number ends in at least r 0s is 2^{-r}

- If there are m different elements, the probability that $R \geq r$ is $1 - \boxed{(\boxed{1 - 2^{-r}})^m}$

Prob. all $h(a)$'s end in fewer than r 0s.

Prob. a given $h(a)$ ends in fewer than r 0s.

Why It Works – (2)

- Note:
- Prob. of NOT finding a tail of length *r* is:

  - If , then prob. tends to 1
    - as

    - So, the probability of finding a tail of length *r* tends to 0

  - If , then prob. tends to 0
    - as
    - So, the probability of finding a tail of length *r* tends to 1
- Thus, will almost always be around *m*.

Why It Doesn't Work

- $E[2^R]$ is actually infinite
  - Probability halves when R $R+1$, but value doubles

- Workaround involves using many hash functions and getting many samples

- How are samples combined?

  - Average? What if one very large value?
  - Median? All values are a power of 2
  - Solution:
    - Partition your samples into small groups
    - Take the average of groups
    - Then take the median of the averages

In-Class Practice 2

- Go to practice

One-Slide Takeaway

- Sampling from a streaming data
  - How to get a fixed proportion or a fixed-size Sample
- Queries over a long sliding windows
  - understand DGIM algorithm
- Filtering Data Streams
  - understand first cut solution and Bloom Filter
- Counting distinct elements
  - Understand Flajolet-Martin Approach
- Appendix: computing moments and counting item sets

References

- Book:
  - Mining of Massive Datasets

- Massive Online Analysis (MOA) Software:

  - http://moa.cms.waikato.ac.nz/

Appendix

- Sampling from a Data Stream
- Queries over a (long) Sliding Windows
- Filtering Data Streams
- Counting Distinct Elements
- Computing Moments
- Counting Itemsets

Generalization: Moments

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

- Let m_a be the number of times value a occurs

- The k^{th} *moment* is

Special Cases

- 0^{th} moment = number of different elements
  - The problem just considered

- 1^{st} moment = count of the numbers of elements = length of the stream

  - Easy to compute

- 2^{nd} moment = *surprise number* = a measure of how uneven the distribution is

Example: Surprise Number

- Stream of length 100; 11 values appear
- Item counts: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9
  Surprise # = 910
- Item counts: 90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1
  Surprise # = 8,110

Due to AMS (Alon, Matias, and Szegedy)

AMS Method

- Works for all moments
- Gives an unbiased estimate

- We'll just concentrate on 2nd moment

- Based on calculation of many random variables *X*:
  - For each random variable *X*, we store *X.el* and *X.val*
  - Each random variable represents one separate item
  - Note this requires a count in main memory, so number of *X*s is limited

One Random Variable

- Assume stream has length n
- Pick a random time to start, so that any time is equally likely
- Let the chosen time have element a in the stream
- $X = n$ * ((twice the number of as in the stream starting at the chosen time) – 1)
  - Note: store n once, count of as for each X

Expected Value of X

- 2^{nd} moment is
- $E(X) = \sum_{\text{all times } t} n * ([\text{twice the number of times the stream element at time } t \text{ appears from that time on}] - 1)$

=

=

Group times by the value seen

Time when the last *a* is seen

Time when the penultimate *a* is seen

Time when the first *a* is seen

Combining Samples

- One random variable only represent one sampled item; we should do many concurrent samples
- Compute as many variables X as can fit in available memory
- Average them in groups
- Take median of averages
- Proper balance of group sizes and number of groups assures not only correct expected value, but expected error goes to 0 as number of samples gets large

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

Stream 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: (fix-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 out, with equal probability

Appendix

- Sampling from a Data Stream
- Queries over a (long) Sliding Windows
- Filtering Data Streams
- Counting Distinct Elements
- Computing Moments
- 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 1s for all items

Extensions

- In principle, you could count frequent pairs or even larger sets the same way
  - One stream per itemset

- Drawbacks:
  - Only approximate
  - Number of itemsets is way too big

Exponentially Decaying Windows

- Exponentially decaying windows: A heuristic for selecting likely frequent itemsets

  – What are "currently" most popular movies?

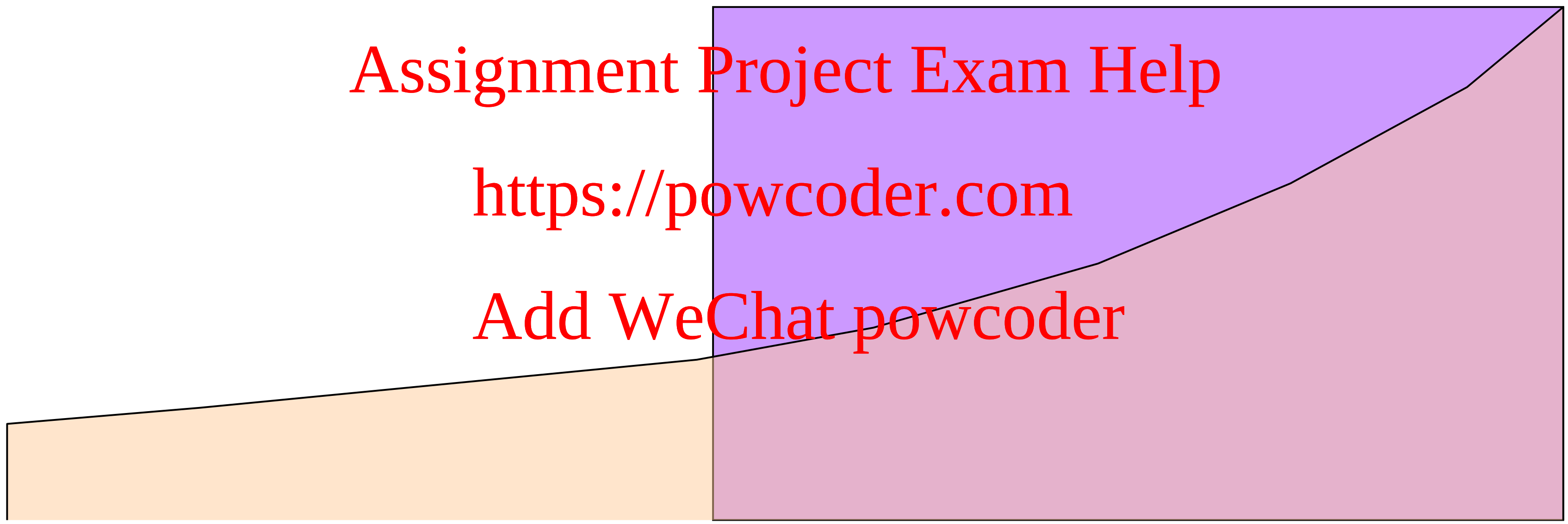
    - Instead of computing the raw count in last *N* elements
    - Compute a smooth aggregation over the whole stream

- If stream is a_1, a_2,… and we are taking the sum of the stream, take the answer at time *t* to be:
  – *c* is a constant, presumably tiny, like or
  – When new arrives: Multiply current sum by *(1-c)* and add

Example: Counting Items

- If each is an "item" we can compute the characteristic function of each possible item *x* as an exponentially decaying window (E.D.W.).
  - That is:
    - where if , and 0 otherwise
  - Imagine that for each item *x* we have a binary stream (1 … *x* appears, 0 … *x* does not appear)
  - New item *x* arrives:
    - Multiply all counts by *(1-c)*
    - Add +1 to count for *x*
- Call this sum the "weight" of item *x*

Sliding Versus Decaying Windows



Important property: Sum over all weights is
=

Counting Items

- Suppose we want to find those items of weight > ½

  - Important property: Sum over all weights is =

- Thus:

  - There cannot be more than items with weight of ½ or more
- So, is a limit on the number of movies being counted at any time

Extension to Larger Itemsets

- Count (some) itemsets in an E.D.W.
  - 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 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 {*i*, *j*} iff both *i* and *j* were counted prior to seeing *B*
  - Start counting {*i*, *j*, *k*} iff {*i*, *j*}, {*i*, *k*}, and {*j*, *k*} were all counted prior to seeing *B*

How Many Counts?

- Counts for single items < (2/*c*) * (average number of items in a basket)

- Counts for larger itemsets = ??

- But we are conservative about starting counts of large sets

  - If we counted every set we saw, one basket of 20 items would initiate 1M counts

In-Class Practice 1

- There are several ways that the bit-stream 10010110111101 could be partitioned into buckets. Find all of them.

In-Class Practice 2

- Suppose our stream consists of the integers 3, 1, 4, 1, 5, 9, 2, 6, 5. Our hash functions will all be of the form for some *a* and *b*. You should treat the result as a 5-bit binary integer. Determine the tail length for each stream element and the resulting estimate of the number of distinct elements if the hash function is: