Mining Data Streams (Part 1)

CS345a: Data Mining
Jure Leskovec and Anand Rajaraman
Stanford University

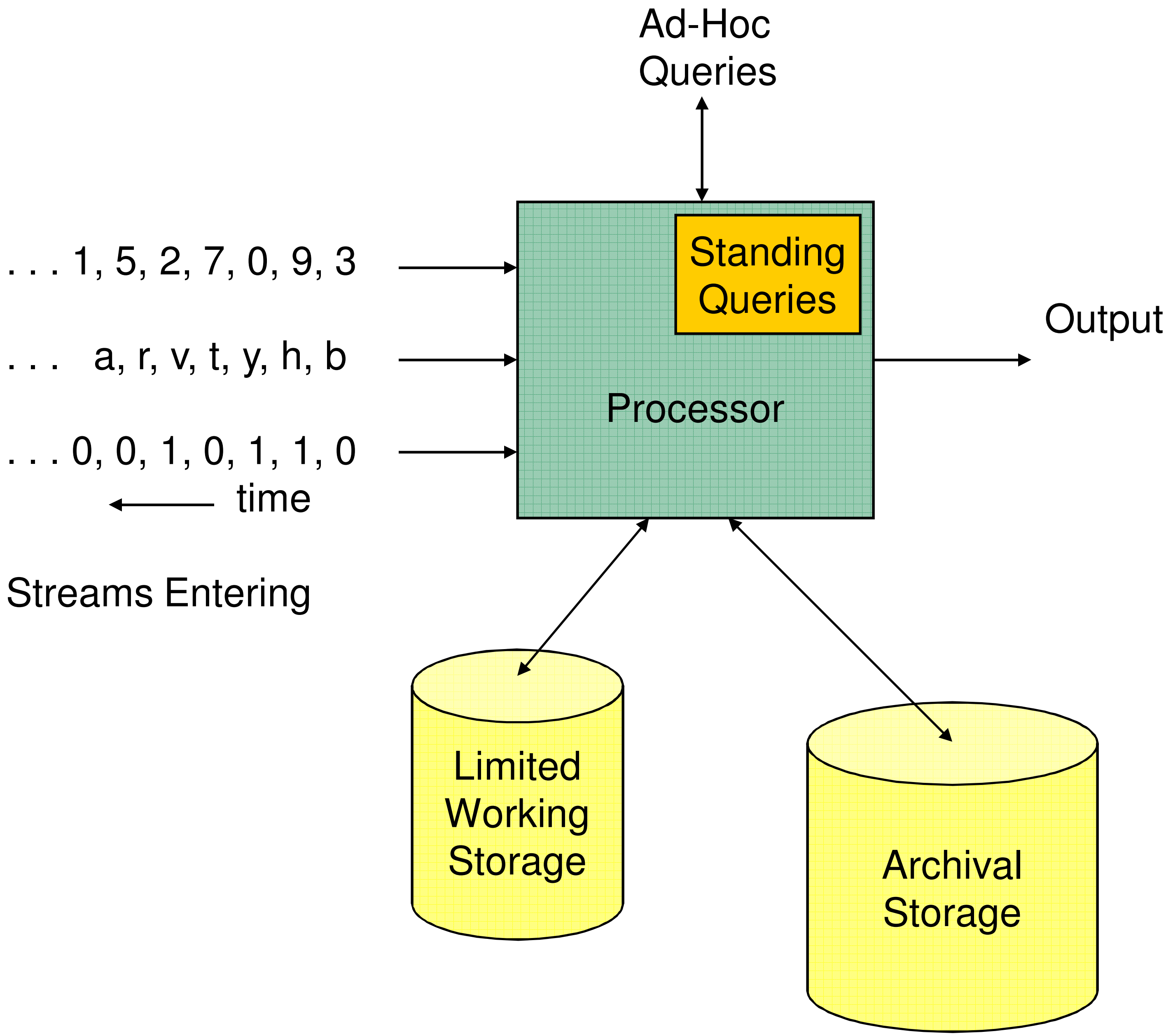
Data Streams

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

- Sometimes the input rate is controlled externally
  - Google queries
  - Twitter or Facebook status updates

The Stream Model

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

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)

- Sensor 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 monitored at a switch
  - Gather information for optimal routing
  - Detect denial-of-service attacks

Data Stream Problems

- Sampling data from a stream
- Filtering a data stream
- Queries over sliding windows
- Counting distinct elements
- Estimating moments
- Finding frequent elements
- Frequent itemsets

Sampling from a Data Stream

- Since we can’t 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

Sampling a fixed proportion

- Scenario: search engine query stream
  - 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
- Naïve solution
  - Generate a random integer in [0..9] for each query
  - Store query if the integer is 0, otherwise discard

Problem with naïve approach

- Consider the 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)
  - Correct answer: $d/(s+2d)$
  - Sample will contain $s/10$ of the singleton queries and $2d/10$ of the duplicate queries at least once
  - But only $d/100$ pairs of duplicates
  - So the sample-based answer is: $d/(10s+20d)$

Solution

- Pick 1/10th of users and take all their searches in the sample
- Use a hash function that hashes the user name or user id uniformly into 10 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*
  - Hash each tuple's key uniformly into *b* buckets
  - Pick the tuple if its hash value is at most *a*

Maintaining a fixed-size sample

- Suppose we need to maintain a sample of size exactly *s*
  - E.g., main memory size constraint
- 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 sample with equal probability *s/n*

Solution

- Store all the first s elements of the stream
- Suppose we have seen n-1 elements, and now the n^{th} element arrives ($n > s$)
  - With probability s/n, pick the n^{th} element, else discard it
  - If we pick the n^{th} element, then it replaces one of the s elements in the sample, picked at random
- Claim: this algorithm maintains a sample with the desired property

Proof: By Induction

- Assume that after *n* elements, the sample contains each element seen so far with probability *s/n*
- When we see element *n+1*, it gets picked with probability *s/(n+1)*
- For elements already in the sample, probability of remaining in the sample is:

$$(1-\frac{s}{n+1})+(\frac{s}{n+1})(\frac{s-1}{s})=\frac{n}{n+1}$$

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.

q w e r t y u i o p 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 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 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 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 1's in the last k bits?" where $k \leq N$.
- Obvious solution: store the most recent N bits.
  - When new bit comes in, discard the $N+1^{st}$ bit.

Counting Bits – (2)

- You can’t 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
- But we’re happy with an approximate answer.

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.

*Datar, Gionis, Indyk, and Motwani

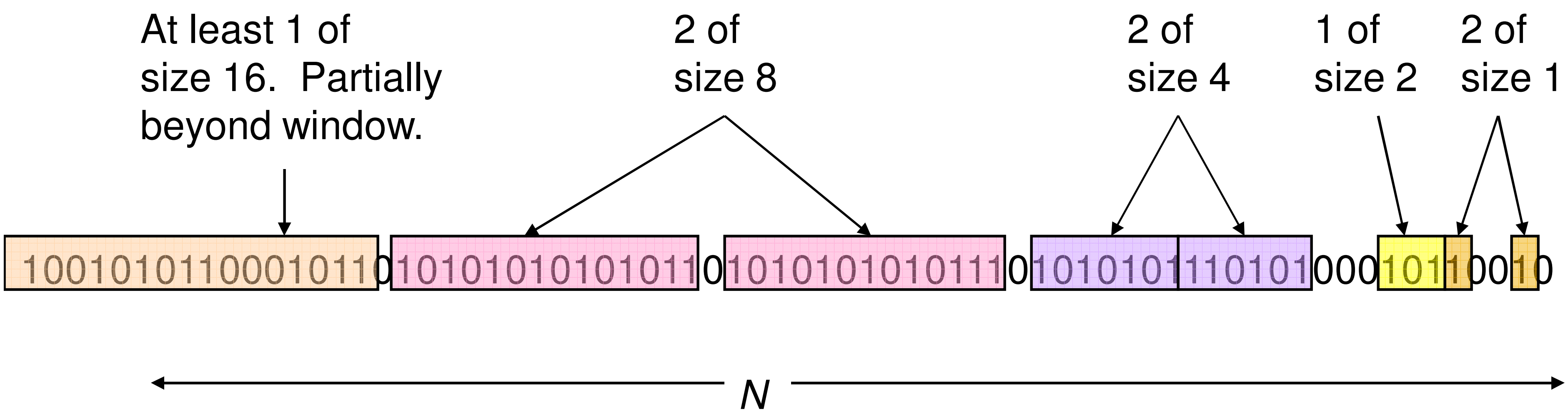
Something That Doesn't (Quite) Work

- Summarize exponentially increasing regions of the stream, looking backward.
- Drop small regions if they begin at the same point as a larger region.

Key Idea

- Summarize blocks of stream with specific numbers of 1's.
- Block *sizes* (number of 1's) increase exponentially as we go back in time

Example: Bucketized Stream



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.

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 1's must be a power of 2.
  - That explains the log log *N* in (2).

Representing a Stream by Buckets

- Either one or two buckets with the same power-of-2 number of 1's.
- 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.

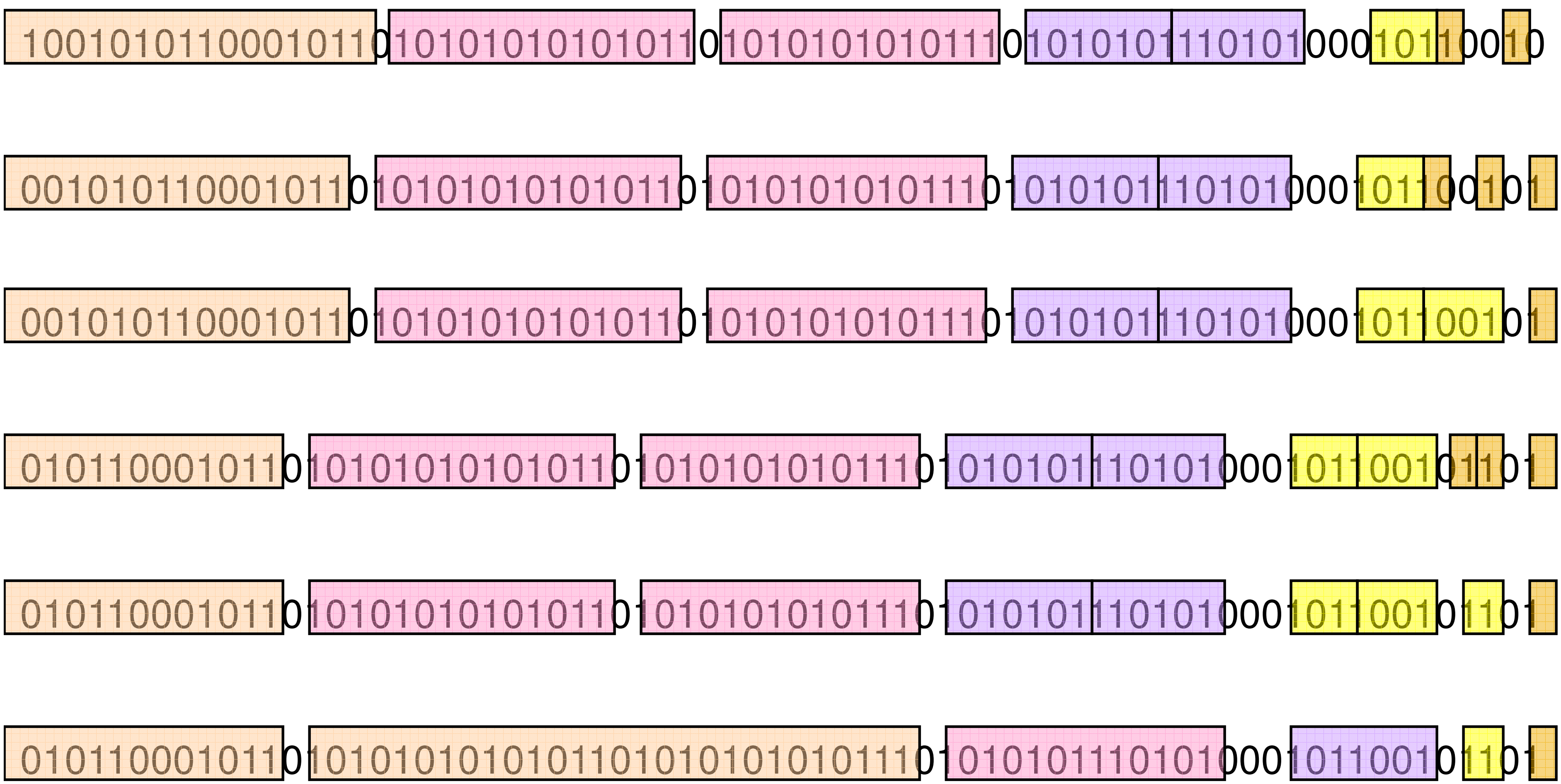
Updating Buckets – (1)

- 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.
- 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

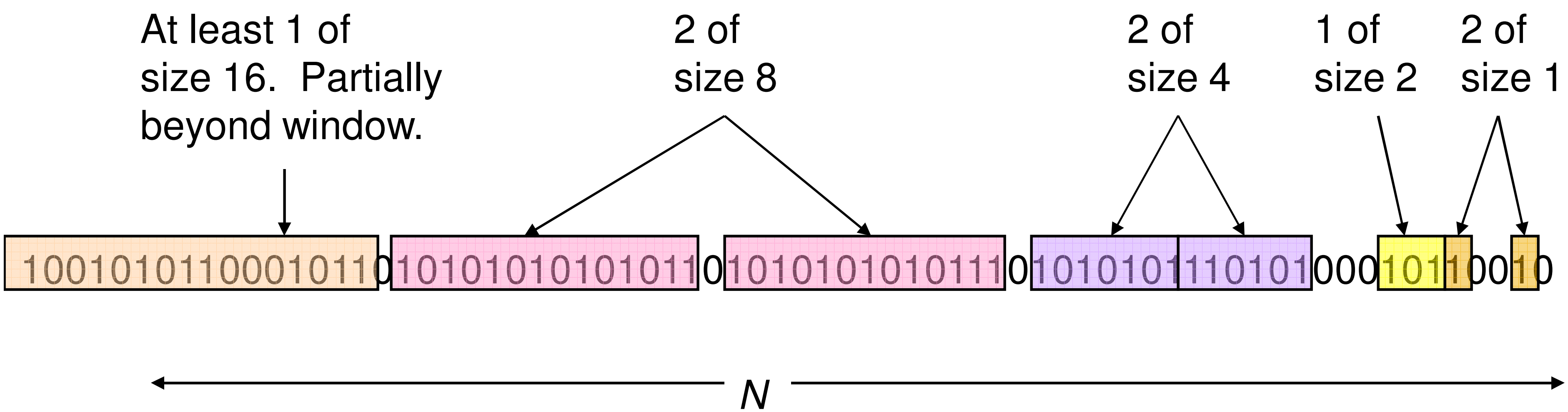

1001010110001011 0 10101010101011 0 10101010101111 0 1010101 110101 000 1011 1 00 1 0
001010110001011 0 10101010101011 0 10101010101110 1010101 110101 000 1011 1 00 1 0 1
001010110001011 0 10101010101011 0 10101010101110 1010101 110101 000 1011 001 0 1
0101100010111 0 10101010101011 0 10101010101110 1010101 110101 000 1011 001 0 1 1 0 1
0101100010111 0 10101010101011 0 10101010101110 1010101 110101 000 1011 001 0 11 0 1
0101100010111 0 1010101010101101010101010111 0 1010101110101 000 1011001 0 11 0 1

Querying

- To estimate the number of 1's in the most recent *N* bits:
  1. Sum the sizes of all buckets but the last.
  2. Add half the size of the last bucket.

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

Example: Bucketized Stream



Error Bound

- Suppose the last bucket has size 2^k.
- Then by assuming 2^{k-1} of its 1's are still within the window, we make an error of at most 2^{k-1}.
- Since there is at least one bucket of each of the sizes less than 2^k, the true sum is at least $1 + 2 + .. + 2^{k-1} = 2^k - 1$.
- Thus, error at most 50%.

Extensions

- Can we use the same trick to answer queries "How many 1's 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 by $1/(r-1)$
- By picking r appropriately, we can tradeoff between number of bits and error